Cramming more components onto integrated circuits

With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip

By Gordon E. Moore

Director, Research and Development Laboratories, Fairchild Semiconductor division of Fairchild Camera and Instrument Corp.

The future of integrated electronics is the future of electronics itself. The advantages of integration will bring about a proliferation of electronics, pushing this science into many new areas.

Integrated circuits will lead to such wonders as home computers—or at least terminals connected to a central computer—automatic controls for automobiles, and personal portable communications equipment. The electronic wristwatch needs only a display to be feasible today.

But the biggest potential lies in the production of large systems. In telephone communications, integrated circuits in digital filters will separate channels on multiplex equipment. Integrated circuits will also switch telephone circuits and perform data processing.

Computers will be more powerful, and will be organized in completely different ways. For example, memories built of integrated electronics may be distributed throughout the machine instead of being concentrated in a central unit. In addition, the improved reliability made possible by integrated circuits will allow the construction of larger processing units. Machines similar to those in existence today will be built at lower costs and with faster turn-around.

The author

Dr. Gordon E. Moore is one of the new breed of electronic engineers, schooled in the physical sciences rather than in electronics. He earned a B.S. degree in chemistry from the University of California and a Ph.D. degree in physical chemistry from the California Institute of Technology. He was one of the founders of Fairchild Semiconductor and has been director of the research and development laboratories since 1959.

Present and future

By integrated electronics, I mean all the various technologies which are referred to as microelectronics today as well as any additional ones that result in electronics functions supplied to the user as irreducible units. These technologies were first investigated in the late 1950's. The object was to miniaturize electronics equipment to include increasingly complex electronic functions in limited space with minimum weight. Several approaches evolved, including microassembly techniques for individual components, thin-film structures and semiconductor integrated circuits.

Each approach evolved rapidly and converged so that each borrowed techniques from another. Many researchers believe the way of the future to be a combination of the various approaches.

The advocates of semiconductor integrated circuitry are already using the improved characteristics of thin-film resistors by applying such films directly to an active semiconductor substrate. Those advocating a technology based upon films are developing sophisticated techniques for the attachment of active semiconductor devices to the passive film arrays.

Both approaches have worked well and are being used in equipment today.

The establishment

Integrated electronics is established today. Its techniques are almost mandatory for new military systems, since the reliability, size and weight required by some of them is achievable only with integration. Such programs as Apollo, for manned moon flight, have demonstrated the reliability of integrated electronics by showing that complete circuit functions are as free from failure as the best individual transistors.

Most companies in the commercial computer field have machines in design or in early production employing integrated electronics. These machines cost less and perform better than those which use "conventional" electronics.

Instruments of various sorts, especially the rapidly increasing numbers employing digital techniques, are starting to use integration because it cuts costs of both manufacture and design.

The use of linear integrated circuitry is still restricted primarily to the military. Such integrated functions are expensive and not available in the variety required to satisfy a major fraction of linear electronics. But the first applications are beginning to appear in commercial electronics, particularly in equipment which needs low-frequency amplifiers of small size.

Reliability counts

In almost every case, integrated electronics has demonstrated high reliability. Even at the present level of production—low compared to that of discrete components—it offers reduced systems cost, and in many systems improved performance has been realized.

Integrated electronics will make electronic techniques more generally available throughout all of society, performing many functions that presently are done inadequately by other techniques or not done at all. The principal advantages will be lower costs and greatly simplified design—payoffs from a ready supply of low-cost functional packages.

For most applications, semiconductor integrated circuits will predominate. Semiconductor devices are the only reasonable candidates presently in existence for the active elements of integrated circuits. Passive semiconductor elements look attractive too, because of their potential for low cost and high reliability, but they can be used only if precision is not a prime requisite.

Silicon is likely to remain the basic material, although others will be of use in specific applications. For example, gallium arsenide will be important in integrated microwave functions. But silicon will predominate at lower frequencies because of the technology which has already evolved around it and its oxide, and because it is an abundant and relatively inexpensive starting material.

Costs and curves

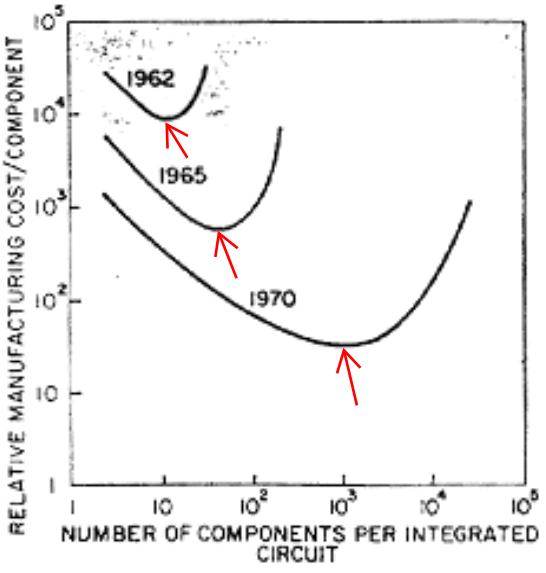
Reduced cost is one of the big attractions of integrated electronics, and the cost advantage continues to increase as the technology evolves toward the production of larger and larger circuit functions on a single semiconductor substrate. For simple circuits, the cost per component is nearly inversely proportional to the number of components, the result of the equivalent piece of semiconductor in the equivalent package containing more components. But as components are added, decreased yields more than compensate for the increased complexity, tending to raise the cost per component. Thus there is a minimum cost at any given time in the evolution of the technology. At present, it is reached when 50 components are used per circuit. But the minimum is rising rapidly while the entire cost curve is falling (see graph below). If we look ahead five years, a plot of costs suggests that the minimum cost per component might be expected in circuits with about 1,000 components per circuit (providing such circuit functions can be produced in moderate quantities.) In 1970, the manufacturing cost per component can be expected to be only a tenth of the present cost.

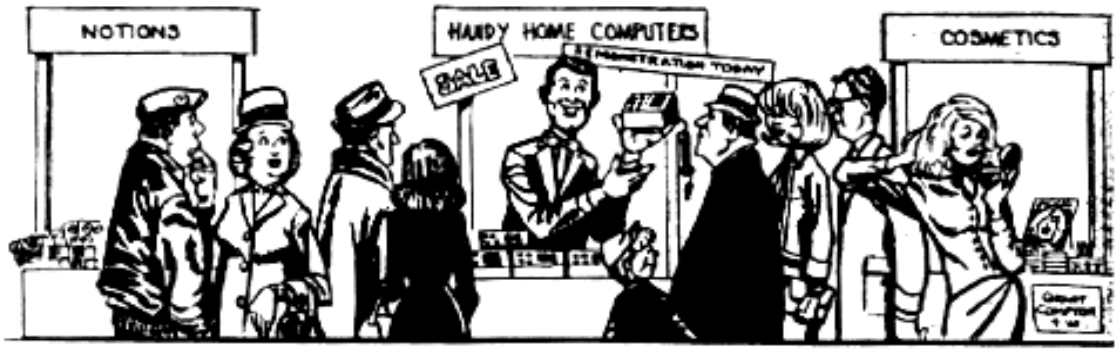
The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (see graph on next page). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000.

I believe that such a large circuit can be built on a single wafer.

Two-mil squares

With the dimensional tolerances already being employed in integrated circuits, isolated high-performance transistors can be built on centers two thousandths of an inch apart. Such



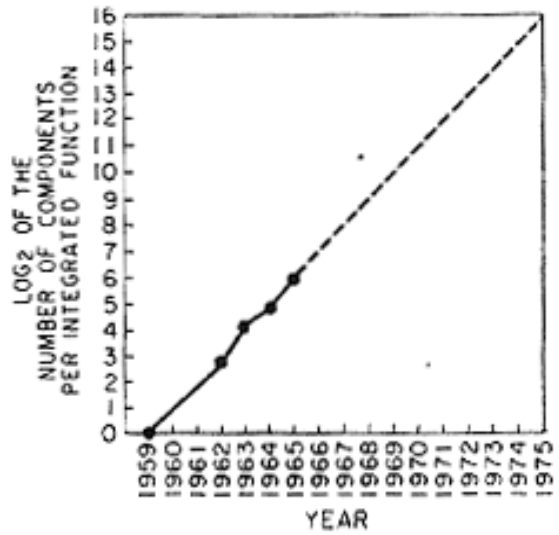


a two-mil square can also contain several kilohms of resistance or a few diodes. This allows at least 500 components per linear inch or a quarter million per square inch. Thus, 65,000 components need occupy only about one-fourth a square inch.

On the silicon wafer currently used, usually an inch or more in diameter, there is ample room for such a structure if the components can be closely packed with no space wasted for interconnection patterns. This is realistic, since efforts to achieve a level of complexity above the presently available integrated circuits are already underway using multilayer metalization patterns separated by dielectric films. Such a density of components can be achieved by present optical techniques and does not require the more exotic techniques, such as electron beam operations, which are being studied to make even smaller structures.

Increasing the yield

There is no fundamental obstacle to achieving device yields of 100%. At present, packaging costs so far exceed the cost of the semiconductor structure itself that there is no incentive to improve yields, but they can be raised as high as is economically justified. No barrier exists comparable to the thermodynamic equilibrium considerations that often limit yields in chemical reactions; it is not even necessary to do any fundamental research or to replace present processes. Only the engineering effort is needed.



In the early days of integrated circuitry, when yields were extremely low, there was such incentive. Today ordinary integrated circuits are made with yields comparable with those obtained for individual semiconductor devices. The same pattern will make larger arrays economical, if other considerations make such arrays desirable.

Heat problem

Will it be possible to remove the heat generated by tens of thousands of components in a single silicon chip?

If we could shrink the volume of a standard high-speed digital computer to that required for the components themselves, we would expect it to glow brightly with present power dissipation. But it won't happen with integrated circuits. Since integrated electronic structures are two-dimensional, they have a surface available for cooling close to each center of heat generation. In addition, power is needed primarily to drive the various lines and capacitances associated with the system. As long as a function is confined to a small area on a wafer, the amount of capacitance which must be driven is distinctly limited. In fact, shrinking dimensions on an integrated structure makes it possible to operate the structure at higher speed for the same power per unit area.

Day of reckoning

Clearly, we will be able to build such component-crammed equipment. Next, we ask under what circumstances we should do it. The total cost of making a particular system function must be minimized. To do so, we could amortize the engineering over several identical items, or evolve flexible techniques for the engineering of large functions so that no disproportionate expense need be borne by a particular array. Perhaps newly devised design automation procedures could translate from logic diagram to technological realization without any special engineering.

It may prove to be more economical to build large

Gordon E. Moore, Co-founder
Intel Corporation

Progress In Digital Integrated Electronics

Complexity of integrated circuits has approximately doubled every year since their introduction. Cost per function has decreased several thousand-fold, while system performance and reliability have been improved dramatically. Many aspects of processing and design technology have contributed to make the manufacture of such functions as complex single chip microprocessors or memory circuits economically feasible. It is possible to analyze the increase in complexity plotted in Figure 1 into different factors that can, in turn, be examined to see what contributions have been important in this development and how they might be expected to continue to evolve. The expected trends can be recombined to see how long exponential growth in complexity can be expected to continue.

A first factor is the area of the integrated structures. Chip areas for some of the largest of the circuits used in constructing Figure 1 are plotted in Figure 2. Here again, the trend follows an exponential quite well, but with significantly lower slope than the complexity curve. Chip area tor maximum complexity has increased by a factor of approximately 20 from the first planar transistor in 1959 to the 16,384-bit charge-coupled device memory chip that corresponds to the point plotted for 1975, while complexity, according to the annual doubling law, should have increased about 65,000-fold. Clearly much of the increased complexity had to result from higher density of components on the chip, rather than from the increased area available through the use of larger chips.

Figure 1 Approximate component count for complex integrated circuits vs. year of Introduction.

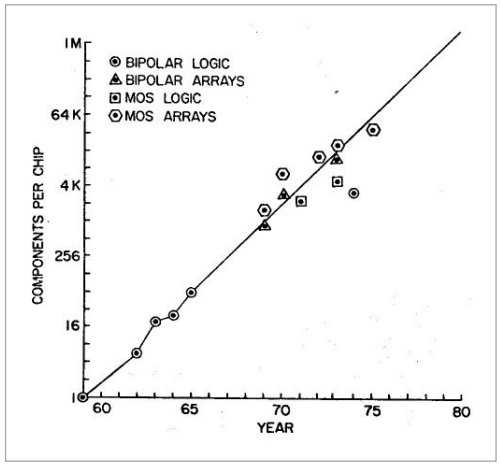
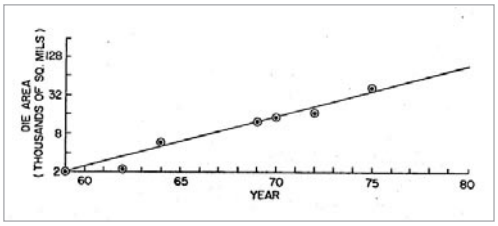


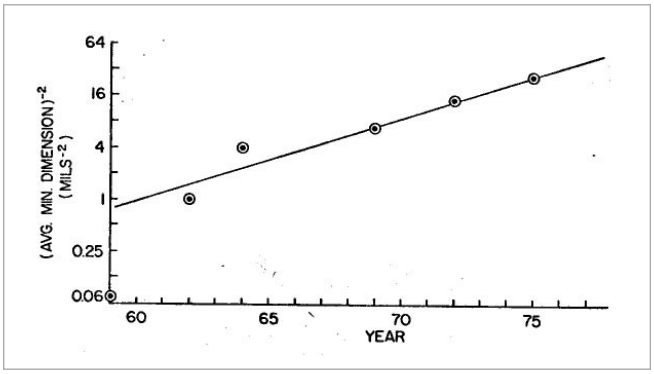
Figure 2 Increase in die area for most complex integrated devices commercially available.



Density was increased partially by using finer scale microstructures. The first integrated circuits of 1961 used line widths of 1 mil (~25 micrometers) while the 1975 device uses 5 micrometer lines. Both line width and spacing between lines are equally important in improving density. Since they have not always been equal,

G. E. Moore, "Progress in Digital Integrated Electronics." © 1975 IEEE. Reprinted, with permission, from Technical Digest 1975. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.

Figure 3 Device density contribution from the decrease in line widths and spacings.

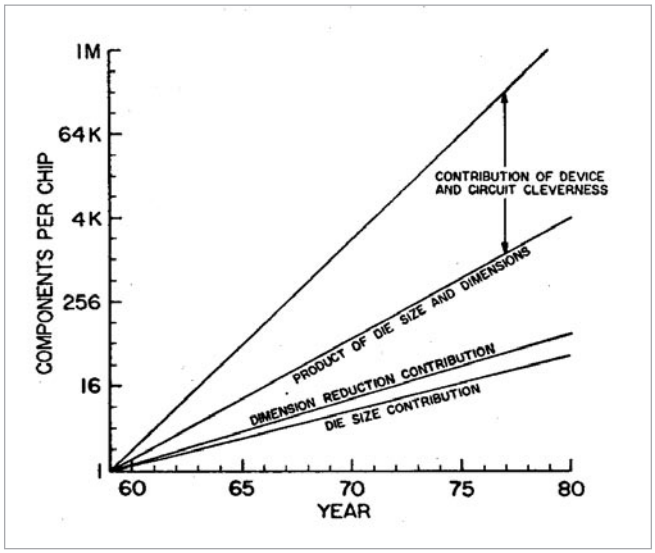


the average of the two is a good parameter to relate to the area that a structure might occupy. Density can be expected to be proportional to the reciprocal of area, so the contribution to improve density vs. time from the use of smaller dimensions is plotted in Figure 3.

Neglecting the first planar transistor, where very conservative line width and spacing was employed, there is again a reasonable fit to an exponential growth. From the exponential approximation represented by the straight line in Figure 3, the increase in density from this source over the 1959-1975 period is a factor of approximately 32.

Combining the contribution of larger chip area and higher density resulting from geometry accounts for a 640-fold increase in complexity, leaving a factor of about 100 to account for through 1975, as is shown graphically in Figure 4. This factor is the contribution of circuit and device advances to higher density. It is noteworthy that this contribution to complexity has been more important than either increased chip area or finer lines. Increasingly the surface areas of the integrated devices have been committed to components rather than to such inactive structures as device isolation and interconnections, and the components themselves have trended toward minimum size, consistent with the dimensional tolerances employed.

Figure 4 Decomposition of the complexity curve into various components.



Can these trends continue?

Extrapolating the curve for die size to 1980 suggests that chip area might be about 90,000 sq. mils, or the equivalent of 0.3 inches square. Such a die size is clearly consistent with the 3 inch wafer presently widely used by the industry. In fact, the size of the wafers themselves have grown about as fast as has die size during the time period under consideration and can be expected to continue to grow. Extension to larger die size depends principally upon the continued reduction in the density of defects. Since the existence of the type of defects that harm integrated circuits is not fundamental, their density can be reduced as long as such reduction has sufficient economic merit to justify the effort. I see sufficient continued merit to expect progress to continue for the next several years. Accordingly, there is no present reason to expect a change in the trend shown in Figure 2.

With respect to dimensions, in these complex devices we are still far from the minimum device sizes limited by such fundamental considerations as the charge on the electron or the atomic structure of matter. Discrete devices with sub-micrometer dimensions show that no basic problems should be expected at least until the average line width and

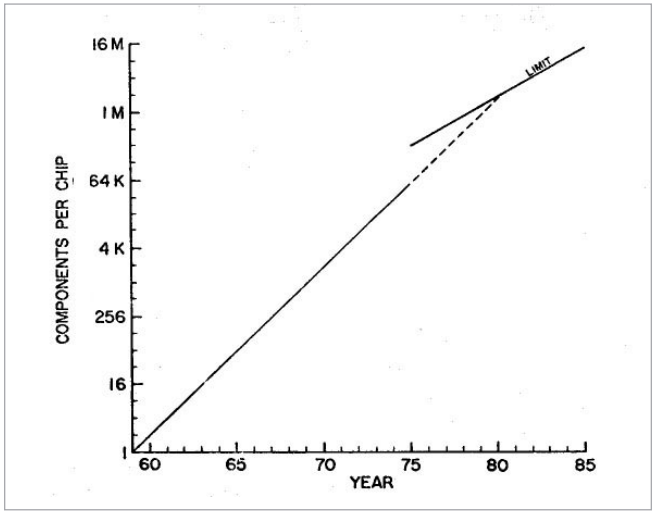
spaces are a micrometer or less. This allows for an additional factor of improvement at least equal to the contribution from the finer geometries of the last fifteen years. Work in non-optical masking techniques, both electron beam and X-ray, suggests that the required resolution capabilities will be available. Much work is required to be sure that defect densities continue to improve as devices are scaled to take advantage of the improved resolution. However, I see no reason to expect the rate of progress in the use of smaller minimum dimensions in complex circuits to decrease in the near future. This contribution should continue along the curve of Figure 3.

With respect to the factor contributed by device and circuit cleverness, however, the situation is different. Here we are approaching a limit that must slow the rate of progress. The CCD structure can approach closely the maximum density practical. This structure requires no contacts to the components within the array, but uses gate electrodes that can be at minimum spacing to transfer charge and information from one location to the next. Some improvement in overall packing efficiency is possible beyond the structure plotted as the 1975 point in Figure 1, but it is unlikely that the packing efficiency alone can contribute as much as a factor of four, and this only in serial data paths. Accordingly, I am Inclined to suggest a limit to the contribution of circuit and device cleverness of another factor of four in component density.

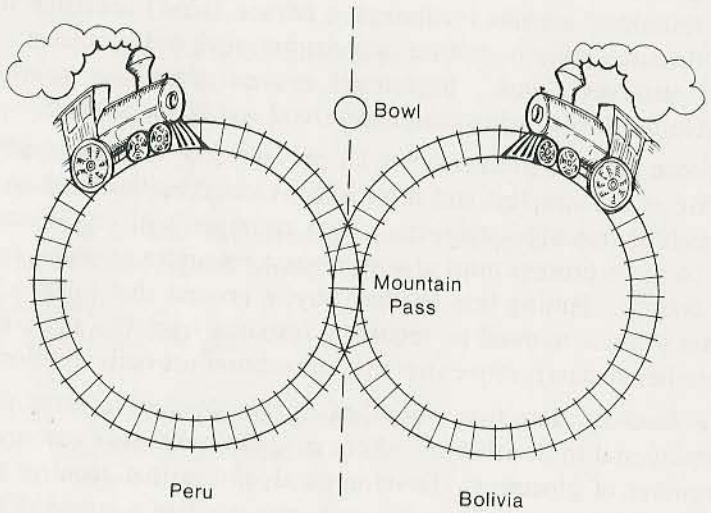
With this factor disappearing as an important contributor, the rate of increase of complexity can be expected to change slope in the next few years as shown in Figure 5. The new slope might approximate a doubling every two years, rather than every year, by the end of the decade.

Even at this reduced slope, integrated structures containing several million components can be expected within ten years. These new devices will continue to reduce the cost of electronic functions and extend the utility of digital electronics more broadly throughout society.

Figure 5 Projection of the complexity curve reflecting the limit on increased density through invention.



14. High in the Andes Mountains, there are two circular railroad lines. As shown in the diagram, one line is in Peru, the other in Bolivia. They share a section of track, where the lines cross a mountain pass that lies on the international border.



Unfortunately, the Peruvian and Bolivian trains occasionally collide when simultaneously entering the critical section of track (the mountain pass). The trouble is, alas, that the drivers of the two trains are blind and deaf, so they can neither see nor hear each other.

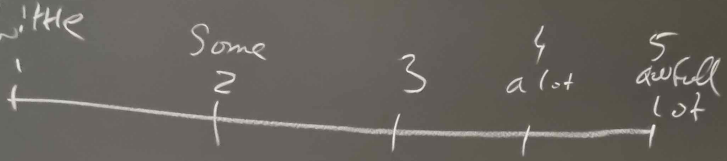
The two drivers agreed on the following method of preventing collisions. They set up a large bowl at the entrance to the pass. Before entering the pass, a driver must stop his train, walk over to the bowl, and reach into it to see if it contains a pebble. If the bowl is empty, the driver finds a pebble and drops it in the bowl, indicating that his train is entering the pass; once his train has cleared the pass, he must walk back to the bowl and remove his pebble, indicating that the pass is no longer being used. Finally he walks back to the train and continues down the line. If a driver arriving at the pass finds a pebble in the bowl, he leaves the pebble there; he repeatedly takes a siesta and re-checks the bowl until he finds it empty.

Then he drops a pebble in the bowl and drives his train into the pass. A smart aleck college graduate from the University at La Paz (Bolivia) claimed that subversive train schedules made up by Peruvian officials could block the Bolivian train forever. (Explain). The Bolivian driver just laughed and said that could not be true because it never happened. (Explain). Unfortunately, one day the two trains crashed. (Explain).

Following the crash, our college graduate was called in as a consultant to ensure that no more crashes would occur. He explained that the bowl was being used in the wrong way. The Bolivian driver must wait at the entry until the bowl is empty, drive through the pass and walk back to put a pebble in the bowl. The Peruvian driver must wait at the entry until the bowl contains a pebble, drive through the pass and walk back to remove the pebble from the bowl. Sure enough, his method prevented crashes. Prior to this arrangement, the Peruvian train ran twice a day and the Bolivian train ran once a day. The Peruvians were very unhappy with the new arrangement. (Why?)

Our college graduate was called in again and was told to prevent crashes while avoiding the problem of his previous method. He suggested that two bowls be used, one for each driver. When a driver reaches the entry, he first drops a pebble in his bowl, then checks the other bowl to see if it is empty. If so, he drives his train through the pass, stops it and walks back to remove his pebble. But if he finds a pebble in the other bowl he goes back to his bowl and removes his pebble. Then he takes a siesta, again drops a pebble in his bowl and re-checks the other bowl, and so on, until he finds the other bowl empty. This method worked fine until late in May, when the two trains were simultaneously blocked at the entry for many siestas. (Explain.)

DID YOU LEARN ANYTHING TODAY



21

OK
38

HARD

1 2+ 11 1

L1 IN3030/IN4330 Lecture 2023-01-25 Summary

- Intro to course, followed slides pretty much including:
  - Why multicore
  - What is caching and why worry about it?
  - Tråde
  - Moore's law
- First half of the lecture was recorded. Unfortunately, the recording of the second half failed ☹ However, the lecture basically followed the slides.

Ukeoppgaver L2 i IN3030/IN4330 – v2023

Eric Jul etter opplegg av *Arne Maus*

Ved gruppetimene vil hjelpelærerne hjelpe med parallelprogrammering i Java. Disse ukeoppgaver er til at øve at lage simple Java programmer med tråde.

1. Hei til verden

Det første alle skal gjøre er å lage et program som genererer like mange tråder k (i tillegg til main-tråden) som det er kjerner på den maskinen du sitter på. Hver tråd skal ha en lokal variabel 'ind'. (ind = 0,1,..,antallTråder-1)

I metoden 'run' skal du plassere først denne setningen:

a) System.out.println("Traad nr: " + ind+" sier hei");

og så følgende to setninger i en try{..} catch blokk:

```
b) try {  Thread.sleep(1000);
          System.out.println("Traad nr: " + ind+" sier hei etter å ha ventet ett sekund") ;
   } catch (Exception e) { return;}
```

Kjør programmet flere ganger – både a) og b) – og se etter om setningene kommer i 'riktig' rekkefølge hver gang. Blandes utskriften?

2. Finn maksimalverdien i en array

Vi skal nå parallellisere problemet med å finne det største elementet i en heltallsarray a[] av lengde n - og skrive: 'int finnMax(int a[]) {..};

Arrayen vi skal først lage en slik array med tilfeldig innhold med klassen Random i pakken: java.util hvor man nytter metoden nextInt(n) for å trekke elementene i a[].

Vi skal nedenfor teste denne for n = 100, 1000, ..., 10 millioner.

a) Lag først en sekvensiell løsning og ta tiden på denne med System.nanoTime() da i nanosekunder og skriv ut tiden i millisekunder. Del nanosekundene med 1000000.0 for å få millisekunder som en double-variabel. Lag en tabell over resultatene.

b) Parallell løsning:
Vi skal så se på to måter å parallellisere dette problemet hvor vi har k tråder:

b.1 Først deler du opp arrayen slik at tråd 0 får de første n/k elementene, tråd 1 har de neste n/k

elementene,.. osv., og den siste tråden får resten av arrayen.

b.2 Deretter deler vi arrayen a[] slik at tråd 0 tester element: 0, k, 2k, 3k,..osv. Tråd 1 tester element nr. 1, k+1, 2k+1, ... Tråd 2 tester element 2, k+2, 2k+2,... Husk å teste slik at man ikke går ut over enden av arrayen.

Vi ser at i begge tilfeller skal trådene oppdatere en felles variabel kalt f.eks. globalMax. Her kan du teste ut to strategier:

b.3 Bruk en synchronized metode som alle trådene kaller for hvert element de har og som oppdaterer globalMax hvis det nye elementet er større enn gammel verdi av globalMax.

b.4 La alle trådene ha hver sin variabel localMax som de bruker til å finne max i sin del av arrayen, Tilslutt kaller hver tråd da bare en gang på en synchronized metode (samme for å sette golbalMax i b.3) som da finner ut hvilken av trådenes lokale max som var størst.

Lag en liten rapport om tidsforbruket med de 4 mulige varianter av algoritmen + tidsforbruket til den sekvensielle algoritmen hvor du også regner ut speedup S. Skriv også om det synes mulig å få S >1 for noen av disse fire alternativene. Kommenter hvorfor noen av algoritmene evt. er spesielt raske eller spesielt treige.

3) Hvis tid: Løs problemet å lage en parallell versjon av summen av elementene i en array

$$Sum = \sum_{i=0}^{a.length-1} a[i]$$

etter samme mal som oppgave 2 . Lag en liten rapport med kjøretidene. Trekker du samme konklusjoner som ved finnMax-problemet?

Sketch about synchronization

Eric Jul and Michael Kirkedal Thomsen

IN3030/IN4330, Spring 2023

Manuscript synchronization sketch IN3030 lecture 8/2-2023 12:15pm at University of Oslo, Large Auditorium, Kristen Nygaards Hus.

1. Michael enters the room several minutes before the start of the lecture – Michael should stand near the blackboard that is the third from the door. Eric enters hurriedly at 10:15:15 and heads for the blackboard second from the door.

2. As soon as Eric enters, both starts writing on each their blackboards (Michael at the 3rd from the door; Eric at the 2nd from the door): “IN3030 Today: Synchronization” and bids welcome to the lecture.

3. Discover each other:

> Eric: "Hi, what are YOU doing here?"
> Michael: "I am doing today’s lecture!”
> Eric: "No, I am doing today’s lecture!”

4. Both pull out a version of the lecture plan:

> Michael (pulls out a printed schedule): ”No, I am doing today’s lecture, see this schedule!”
> Eric (pulls out a printed schedule): ”No, I am doing today’s lecture, see THIS schedule!”
> Michael: ”Mine is the REVISED schedule!” (points to the word REVISED)
> Eric: ”Mine also says REVISED!” (points to the word REVISED)

5. EJ: "What do we do about this?”

6. Short pause, then: both *in unison* (triggered by Eric sweeping his hand across toward Michael): "After you!!!" (repeat 3 times, *i.e.*, a total of 4 times).

7. Eric: "This is not working out!! How are we to decide who should do the lecture?? I must be able to devise a method for us to decide who should lecture – after all, I AM a PROFESSOR!” (Eric scratches his head…) – ”… hey, I have a great idea! A FLAWLESS system that will ensure that there is only ONE lecturer!”

8. Eric: Sets up an opaque bowl and explains: “Here is how we ensure that there is no more than one lecturer: When entering, a lecturer goes over to the bowl and check whether or not the bowl is empty. If empty, the lecturer then finds a piece of chalk and puts it into the bowl, indicating that there now *is* a lecturer. If NOT empty then the lecturer knows that there ALREADY is another lecturer, and thus can leave. This FLAWLESS system thus *ensures* that there is ONLY one lecturer!”

9. Eric then walks over to the bowl, checks that it is empty then moves away gets a piece of chalk. He then starts bragging about his FLAWLESS system to ensure that only one will be lecturing while waving the piece of chalk in his hand. As Eric is walking away from the bowl, Michael resolutely and quickly moves over to the bowl, checks it (flip it upside-down), then swiftly finds a piece of chalk, dumps it into the bowl and then moves to the blackboard away from the door and starts writing the agenda on the board.

10. When Eric is done bragging about his FLAWLESS system, he goes to the bowl and drops the piece of chalk into the bowl and then goes back to the board. BOTH start lecturing – saying welcome and that today, we will talk about *synchronization* – BOTH stop when discovering that there are **TWO** lecturers.

11. Small discussion: Michael (sarcastic, shaking head): “FLAWLESS system indeed!”.

12. Eric (again scratching his head): “Let me think … Ah! Now I know: I need TWO bowls. And now we start by putting chalk into one’s OWN bowl and then checking the OTHER bowl. Eric: "This method ensures that there is AT MOST ONE LECTURER!”

13. Both now go to the blackboard, find a piece of chalk, put it into one’s own bowl, then goes over to the other bowl to find chalk in it. Then both points to the other and says: “Ah, it is YOU that will lecture today!” and then both pack up and leave.

14. Eric returns and starts the actual lecturing.

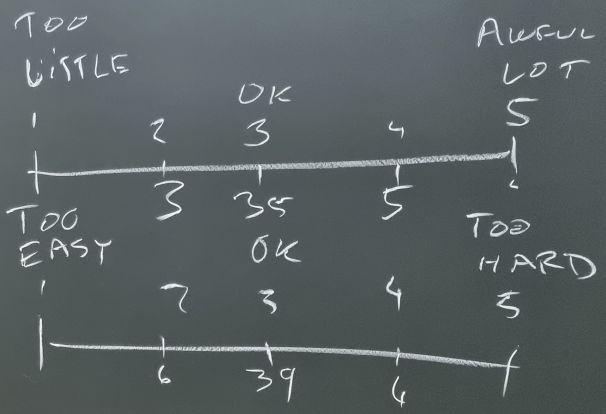
Assignment: (Think carefully about your answer – and explain, *e.g.*, with examples):

- **What went wrong in point 4, 6, 9-10, and 12? Suggest a fix for the problem in 6.**
- **Is the statement at the end of point 8 correct?**
- **Is the statement at the end of point 12 correct? Why is it not sufficient?**

EVAL

LEARN

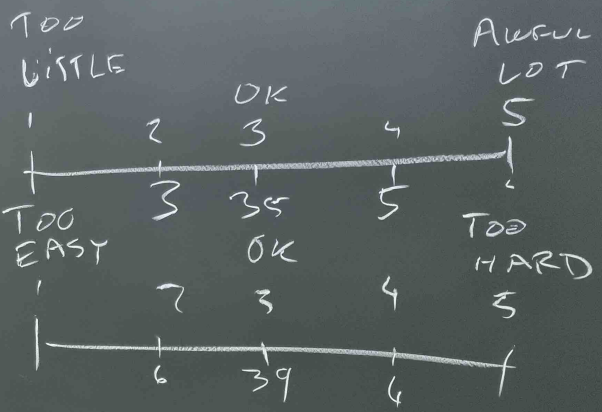
HARD



EVAL

LEARN

HARD



February 15th, 2023

Oblig 2 IN3030/IN4330 – v2023
Parallelization of Matrix Multiplication

This compulsory assignment (Oblig) is about Matrix Multiplication.

Given two matrices `A` and `B` of size `N` x `N`, you are to implement and run three variants of Matrix Multiplication `A x B`: one using the classic algorithm, one where `B` is transposed, and one where `A` Is transposed. For each version, you should implement and run both a sequential and a parallel version of Matrix Multiplication and measure and report on the speedup achieved.

You are to run with matrix sizes of `100x100`, `200x200`, `500x500`, and `1000x1000`.

Your program must be able to run regardless of the number of cores and should utilize them all by default; you can also optionally set a maximum number of cores as a parameter to the program.

In your solution include a check that you get the correct results, based on the sequential algorithm. Beware, that due to rounding errors, the results might differ very slightly—for floating point numbers `(a+b)+c` may not be exactly the same as `a+(b+c)`.

Show the speedup results both in table form and show a graphical representation of the speedup results.
You should achieve a speedup for at least some of the larger matrices.
Furthermore, show a graph comparing all six variant – use the variant that is the fastest (for large `N`) as a baseline, so that the others are depicted relative to the fastest.

You must use the published precode that we provide to fill the two arrays to be multiplied.

Be aware that the sequential code can be slow, especially for larger matrices.

You must write a short report explaining how you did the parallelization, how you synchronized and why (or why not) you achieved a speedup. Comment on why the speedup varies with different sizes of the matrices. Include the table and graphs in the report. It should also contain a short user's guide explaining how to execute your code. The submitted code must be executable by following the user's guide.
The report is to be delivered in PDF format. Your report, the Java code and any other file are to be uploaded as a single zip file or tar file. The uploaded code must be compilable and runnable. At least some of your speedups should be greater than one.

For IN4330 students: You must, in addition, run your program on two *different* machines, *e.g.*, your own laptop and one of Ifi's machines. Thus, the results and graphs are in two versions: one set for each machine. Comment on the difference.

Note: for the highly skilled parallel programmer – there are even better methods than this one, *e.g.*, *tiling* but we are requesting that you do the simpler transposition solution described above ;-)

Tiling will be discussed briefly in a later lecture.

Delivery: Deadline is February 28th, 2023 at 23:59:00 CET – that is 23:59:00 local time in Oslo.

Deliver in devilry.

Template for report:

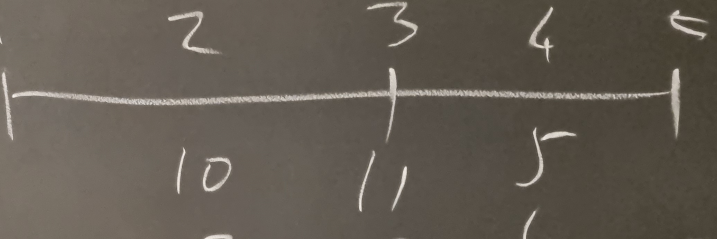
1. Introduction
2. Sequential Matrix Multiplication – *short description*
3. Parallel Matrix Multiplication – *how you did the parallelization*
4. Measurements – *includes discussion, tables, graphs*
5. User guide – *how to run your program*
6. Conclusion – *just a short summary*

Appendix:

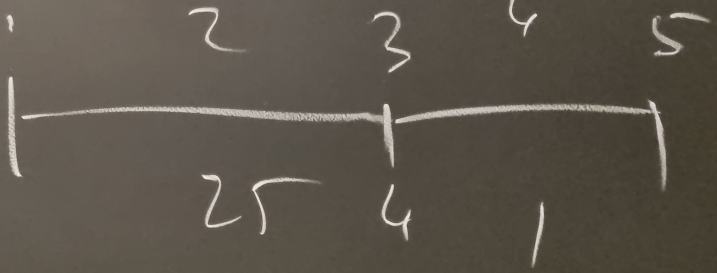
- Your Java code
- Any supporting files, *e.g.*, makefile, whatever …

EVAL

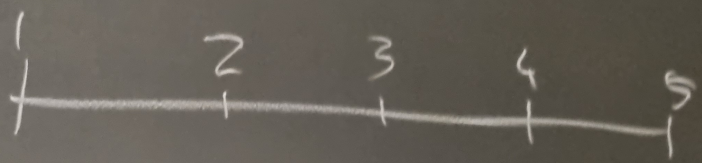
LEARN



HARD



EVAL



Learn 10 32 1

HARD 5 16 5

532 266 133 19

2 266 2 133 7 19 19

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~

p

Step: $2 \times p$ Start at p^2

2 3 5 7 11 13 17 19 23

To factorise k

Need primes up to $\sqrt{k}$

To generate primes $< \sqrt{k}$ need sieve upto: $\sqrt{\sqrt{k}} = \sqrt[4]{k^2}$ $k^2 < N$

$k < \sqrt{N}$

$k^2 < (\sqrt{N})^2$

$10^{12} \sim 10^3$

Oblig3 IN3030/IN4330 v2023: Prime Numbers

Eric Jul, February 8th, 2023

This is mandatory assignment number 3 (Oblig3) for IN3030/IN4330, Spring 2023. The subject is calculation of prime numbers and the unique factorization of a given number into the product of only primes.

You are to take the Java sequential version of the sieve of Eratosthenes that will be published on the IN3030 web site and produce a Java parallel version that achieves speedups greater than one. The program must also sequentially calculate the factorization of large numbers based on the primes that your program generates.

Your program must contain both the sequential and a parallel version of the sieve of Eratosthenes and of the factorization of large numbers. The program must take a parameter, N, greater than 16, and then first generate all the primes less than N, and thereafter calculate the factorization of the 100 largest numbers less than N * N. You must print the factorizations using the precode that we post on the web. Do not modify the precode. Simply place the file in the same folder as your own code, and use it as shown in the comments of the source file.

The program must also take a second parameter, k, the number of threads to be used, if zero then the program uses the number of cores on the machine. Note that your program must contain both the sequential and a parallel version of the sieve of Eratosthenes and both the sequential and a parallel version of factorizing a single number. You must test that your program (in both the sequential version and the parallel version) correctly calculates the primes less than 100 by printing the primes and manually checking them. You must run your program in the full version that finds all primes up to N and factorizes the largest 100 numbers less than N*N for the following four values of N: 2 million, 20 million, 200 million, 2 billion (note: 2 milliarder in Norwegian). For each run, you must print the execution times and speedups obtained. In the report, you must include the median times for a set of runs (at *least* 7 runs) and speedups achieved when generating primes and when factorizing the top 100 numbers. The 100 factorizations must be printed to a file using a class (our precode) that will be uploaded to the course web site. The order of the factors must be monotonically increasing. You MUST parallelize EACH factorization, i.e., use multiple threads even for a single factorization. You program should work for any number of cores.

Further requirements:

- Your program must *test* that the sequential and the parallel versions generate the SAME prime numbers.
- You must include two graphs of the speedup, one for each algorithm, where the X-axis is the four values of N (perhaps log-scale), and the Y-axis is the speedup.
- The execution time for the sieve must not exceed 30 seconds - if it does, you are doing something wrong. Similarly, the time for factorization should not exceed 60 seconds.
- When printing timing values make sure to print the correct unit, *e.g.*, s, ms, ns, or whatever SI-unit you use.
- The program should print good error messages, if it is given incorrect/missing parameters.
- Use good programming style, *e.g.*, proper indentations, good variable names, *etc*.
- **IN4330 students** also have to: give more careful analysis of data obtained using Java Benchmarking Harness.

The report should contain:

1. Introduction – *what this report is about.*
2. User guide – *how to run your program (short, but essential), include a very simple example.*
3. Parallel Sieve of Eratosthenes – *how you did the parallelization – consider including drawings.*
4. Parallel factorization of a large number – *how you did the parallelization – consider including drawings.*
5. Implementation – *a somewhat detailed description of how your Java program works & how tested.*
6. Measurements – *includes discussion, tables, graphs of speedups for the four values of N, number of cores used.*
7. Conclusion – *just a short summary of what you have achieved.*
8. Appendix – *the output of your program.*

Deadline

Deliver in devilry: Deadline is April 11th, 2023 at 23:59 CET – that is 23:59:00 local time in Oslo.

UNIVERSITETET I OSLO

Det matematisk-naturvitenskapelige fakultet

Eksamen i:	INF2440— Effektiv parallellprogrammering
Eksamensdag:	7. juni 2016
Tidspunkter:	09.00 – 13.00
Oppgavesettet er på:	3 sider + 1 side vedlegg
Vedlegg:	I) Skisse av Modell2-koden med litt fra oppgave 3,
Tillatte hjelpemidler:	Alle trykte og skrevne notater, utskrifter, bøker ol.

- Kontroller at oppgavesettet er komplett, og les nøye gjennom oppgavene før du løser dem. Poengangivelsen øverst i hver oppgave angir maksimalt antall poeng.
- Du kan legge dine egne forutsetninger til grunn og gjøre rimelige antagelser, så lenge de ikke bryter med oppgavens "ånd". Gjør i så fall rede for disse forutsetningene og antagelsene.
- Til eksamen skal svarene skrives på gjennomslagspapir. Da må du huske å skrive hardt nok til at besvarelsen blir mulig å lese på alle gjennomslagsarkene, og ikke legge andre deler av eksamensoppgaven under når du skriver.
- Til eksamen skal du selv beholde underste arket etter levering av de to øverste til eksamensinspektøren. Nummerér sidene, og husk å skrive kandidatnummeret ditt på besvarelsen.

I vedlegg I) finner du en litt forenklet versjon av Modell2-koden som ble nyttet i kurset (en ytre klasse med main-tråden og en indre klasse hvor objekter av denne blir egne tråder) I den skissen er det markert med store bokstaver ulike områder av denne koden som det blir referert til i oppgavene.

Oppgave 1 (10 poeng)

a) Beskriv meget kortfattet de to viktigste egenskapene ved tråder i et Java-program.

b) Terminerer programmet alltid når maintråden er ferdig? Begrunn svaret.

Oppgave 2 (20 poeng)

a) Tenk deg et program hvor det er en tråd som usynkronisert både leser og øker en variabel 'x' (initielt = 0) med 1 hver gang den er ferdig med en deloppgave, mens de andre trådene går i løkke, leser 'x' og skal terminere når **x == 20**. Kan dette gå galt – vil alle 'lese-trådene' sikkert terminere hvis det løses minst 22 slike deloppgaver? Begrunn svaret (maksimalt 10 linjer).

b) Tenk deg et program hvor det er en tråd som usynkronisert både leser og øker en variabel 'x' (initielt = 0) med 1 hver gang den er ferdig med en deloppgave, mens de andre trådene går i løkke, leser 'x' og skal terminere når **x ≥ 20**. Kan dette gå galt – vil

alle leser-trådene sikkert terminere hvis x økes til minst 22? Begrunn svaret (maksimalt 10 linjer).

Oppgave 3 (25 poeng)

Se på koden i Vedlegg 1, der vi har et lite parallelt program som skriver litt ut i main og i run. Merk den litt uvanlige initiering av CyclicBarrier b i forhold til hvor mange tråder vi har.

a) Hva skriver programmet ut (hvor mange A-er og hvor mange B-er og kommer teksten 'Main terminerer' ut)?
b) Terminerer programmet – begrunn svaret.
c) Hvis du endrer **`num =3`** til **`num=2`**, hva skriver programmet ut da?
d) Hvis **`num=2`** som i pkt. c), terminerer da programmet – begrunn svaret
e) Som du ser står metoden :**`'void sync()`**' deklarert i området B, lokalt i Arbeider-trådene . Tenkt deg at du flyttet denne deklarasjonen opp til området A i den ytre klassen. Beskriv kort hva dette vil ha å si for oppførselen av programmet i pkt. a)-d).

Oppgave 4 (50 poeng)

Vi antar at du har løst Oblig2 med Erathostenes Sil. Vi er interessert i å teste om det **siste desimale siffer** i alle primtall under 1 milliard er jevnt fordelt (dvs. om det er jevnt over like mange 1,3,7 og 9-ere som er siste siffer i primtallene) . Det er opplagt at det er bare to primtall som slutter på 2 og 5, nemlig tallene 2 og 5 selv, og ingen andre primtall slutter på 0,2,4,6 og 8 (fordi slike tall er delbare på 2).

Du lager en Erathosthenes sil slik:

```
ErathosthesSil s = new ErathosthesSil(1000000000);
```

Vi antar at du kaller denne bare en gang og nytter den sekvensielle Silen du laget i Oblig2. i både punktene a) og b) nedenfor.

Erathosthes Sil inneholder en metode:

```
int nextPrime(int i) {
  // returnerer neste primtall >'i'
     return  ..;
}
```

Denne skal du bruke til å undersøke påstanden at siste siffer i primtallene er meget jevnt fordelt (primtallsforskerne undersøker virkelig dette nå).

Oppgave 4.a) Skriv et sekvensielt program som lager en tabell over hvor mange av primtallene > 10 som ender på 1,3,7 eller 9. Bruk da silen du allerede har laget og metoden **`int nextPrime(int i)`** (som du ikke skal skrive, men anta er riktig og bare kan bruke). Beregn også det siste sifferet som færrest primtall slutter på i prosent av det siste sifferet som flest primtall slutter på. Skriv formelen for svaret.

Oppgave 4.b) Skriv et parallelt program som finner samme tabell som i oppgave 4.a. (Du skal altså bare parallellisere det å finne denne tabellen – ikke det å lage Erathosenes Sil eller det å finne den største og minste av 4 tall). Bruk Modell2-koden og forklar bare hvor det du skriver skal plasseres inn der.

Oppgave 5 (40 poeng)

Skriv et sekvensielt og parallelt program som søker etter og finner ut om, og eventuelt da hvor, et tall 's' er i en usortert array: **`int [] a = new int[n].`** Det er mulig at 's', det tallet du leter etter, finnes flere steder i arrayen, og da er det likegyldig hvilke av stedene du svarer.
Du skal bruke Modell2 koden i vedlegget og bare skrive én metode (med nok parametre) som søker i arrayen og som nyttes av både den sekvensielle løsningen og den parallelle løsningen. I tillegg i den parallelle løsningen kan det komme kode run() metoden som sammenligner svaret fra de k trådene du starter, men det er kanskje ikke tilfellet her? Begrunn valget ditt på dette punktet.
Svaret skal foreligge enten som **-1** i en globalt synlig variabel **`int svarIndeks`**; (i området A i vedlegget) som betyr at tallet ikke fantes; eller som et tall **>= 0** i **`svarIndeks`** som da sier en plass i **`a[]`** du finner tallet 's' som du leter etter. Gi også en vurdering av om denne oppgaven følger Amdahl lov eller Gustavsons lov når vi øker n, lengden av a[].

Oppgave 6 (60 poeng)

Du har en foreleser som tror at han har en genial idé til å lage en raskere innstikksortering-metode kalt swapSort. Problemet med innstikksortering, tenkte han, er at det er en del store elementer tidlig (med lave indekser) i arrayen som må skyves langt avgårde mot enden av arrayen, og motsatt mange små verdier alt for langt ut i array-en som sorteres mot begynnelsen. Disse må flyttes (byttes med hverandre) før vi gjør innstikksortering til sist.

a) Hvis vi sammenligner alle par av to elementer (a[i] og a[n/2+i], i = 0,1,…n/2-1) mot hverandre og bytter om de to hvis den som er til venstre er større enn den til høyre, så vil arrayen bli mye raskere å sortere med innstikksortering etterpå.
b) Ved nærmere ettertanke, tenkte foreleseren din, hvis dette er lurt, så kan vi rekursivt gjenta det, først for hel arrayen, så for første halvdel og så for halvdel rekursivt hver for seg, og igjen for disse halvdelenes halvdeler igjen så lenge lengden av det området vi skal bytte om på er > 10. Denne rekursjonen utfører du altså etter at du har swapSortert hele arrayen .
c) Når vi er ferdige med alle disse ombyttingene, gjøres ett kall på innstikksortering av hele arrayen.

Oppgave:
Bruk Modell2-koden og forklar bare hvor det du skriver nedenfor skal plasseres inn der.

1. Programmér denne algoritmen sekvensielt.
2. Lag en parallell versjon av dette sekvensielle programmet (bare parallelliser ombyttingene – ikke det avsluttende kallet på innstikksortering).

Du kan anta at du har gitt en metode:

```
void innstikkSort(int[] a, int left, int right)
```

som sorterer arrayen a[] fra og med element a[left] til og med element a[right].

Til sist : Gi din vurdering om dette er en god idé eller ganske dårlig idé – begrunn svaret.

Appendix I – Modell2 kodeskisse med litt fra oppgave 3:

```
import java.util.concurrent.*;

class Problem {
    //  felles data og metoder A
     static int num = 3;                                  // MERK – oppg.3
     CyclicBarrier b = new CyclicBarrier(num);    // MERK – oppgave3

     public static void main(String [] args) {
          Problem p = new Problem();
          p.utfoer(num+1);                                // MERK – oppg. 3
          System.out.println(" Main-tråden  TERMINERER"); // MERK – oppg. 3
     } // end main

     void utfoer (int antT) {
          Thread [] t = new Thread [antT];
          for (int i =0; i< antT; i++)
              ( t[i] = new Thread(new Arbeider(i))).start();
           try{
             for (int i =0; i< antT; i++) t[i].join();
          } catch(Exception e) {}
     } // end utfoer

     class Arbeider implements Runnable {
          // lokale data og metoder B
          void sync() { try{ b.await(); }  // MERK – oppgave3
                       catch (Exception e) { return;}
           }
          int ind;

          Arbeider (int in) {ind = in;}

           public void run( ) {
                  // kalles når tråden er startet –  MERK oppg.3
                    sync();
                    System.out.println("A");
                    sync();
                    System.out.println("B");
            } // end run
     } // end indre klasse Arbeider
} // end class Problem
```

Oblig 4 IN3030/IN4330, Spring 2023 Finding the Convex Hull of a set of Points in a Plane: A Recursive Geometric Problem

Delivery deadline: May 3rd 2023, 23:59:00 CEST (local Oslo time)

Original version written by Arne Maus, this version edited by Eric Jul

Given a set of points P in a plane consisting of n (n >2) different points pi (i =0,1,2,…,n-1) with integer coordinates (i.e., no decimals), where we can assume that *not all* points are on the same line, find the Convex Hull of the set of points. The convex is a polygon consisting of a subset of the points in P and with lines between some of these points such that all the other points are on the inside of this polygon and also so that all inner angles of this polygon are ≤ 180. In simpler terms convex means that there is no inwards dent on the set of lines running around the set of points (fig1).

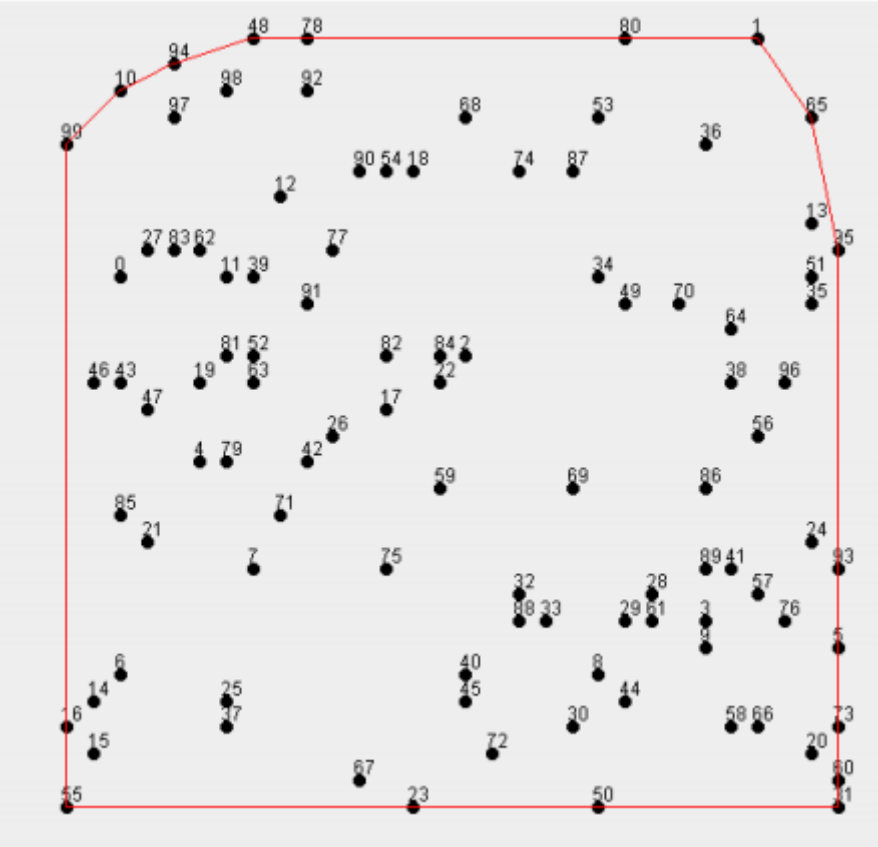


Fig1: The Convex Hull of 100 arbitrary points in a plane (the same as used in the assignment).

When finding such a convex hull, we start with an arbitrary point along the hull and jot down the following points that make up the hull *counter-clockwise*. In fig1, it could for instance be 16, 55, 23, 50, 31, 60, 73, 5, 93, 95, 65, 1, 80, 78, 48, 94, 10, 99. Notice that if there are more points on a line (for instance, 31, 60, 73, 5, 93, and 95), all of those points should be included in the hull. You are in other words not allowed to go straight from point 31 to 95, you have to specify all the lines 31-60, 60-73, 73-5, 5-93 and 93-95. We thus require that all points in P that are on one of the lines of the polygon, are part of the hull. This makes the subset of points included in the convex hull unique.
What point should you start with? Note that any point that has either the highest or lowest x or y value will always be on the hull, so any such point can be used as a starting point. For example points 31 or 55 in fig1.

How do we find the Convex Hull of n points?

We can start with two points that we can see are obviously on the convex hull: the point that has the lowest value for x and the point that has the highest value for x. If there are more points with the same values, just arbitrarily choose one of those with the lowest and one of those with the highest values for x (in the example, for example 16 and 5). The rest of the algorithm is based on simple geometry, which is reviewed below.

Equation for a line

Each line from a point p1 (x1, y1) to p2 (x2, y2) can be written as:

$$\mathbf{a}x + \mathbf{b}y + \mathbf{c} = 0$$

Where: **a** = y1 − y2, **b** = x2 − x1 and **c** = y2 ∗ x1 − y1 ∗ x2

Notice that this is a **straight line *from* p1 *to* p2.**

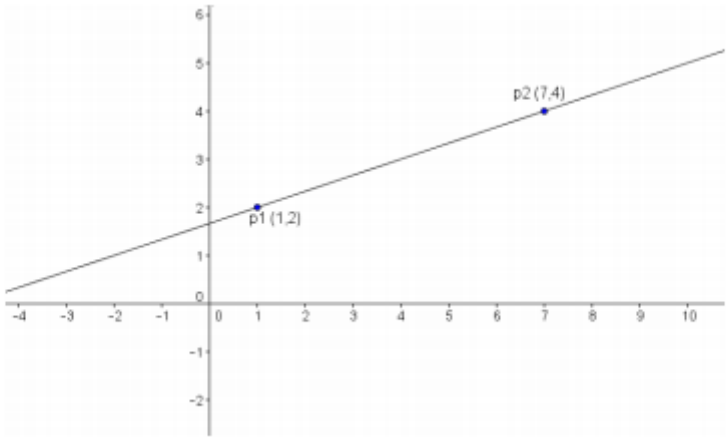


Fig2: A line from p1 (1,2) to p2 (7,4) has the line equation:
(2 − 4)x + (7 − 1)y + (4 ∗ 1 − 2 ∗ 7) = 0; i.e. − 2x + 6y − 10 = 0

The line equation means that all points on this line satisfy the equation. We can also say that this line splits the plane in two: the points to the left side seen from the direction of the line - from p1 to p2 - and the points on the right side in the direction from p1 to p2.

The distance from the line to other points

If we consider points that are not on the line p1-p2, we can see that all points, for example p3 (5,1) on the right side of the line gives a number <0 in the line equation ax + by + c, og all points to the left of the line, such as p4 (3,6) gives a number >0.

The further from the line the points are, the larger negatives or positive values they will have. Generally speaking we can say that the distance perpendicular down from a point p (x,y) to a line ax + by + c is:

$$d = \frac{ax + by + c}{\sqrt{a^2 + b^2}}$$

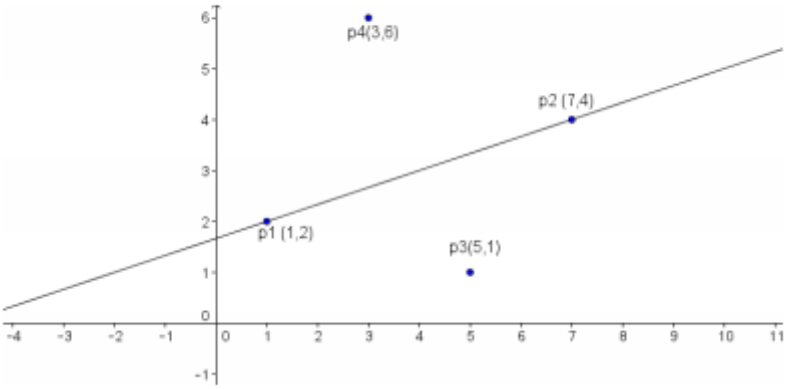


Fig3: The distance from a point to a line. The line divides the plane in two; those points with negative distance (to the right) and those with positive distance (to the left) from the line.

Consider two points in fig3: p3 (5,1) and p4 (3,6), we can calculate the distance from p4 to the line p1-p2 as:

$$\frac{-2*3+6*6-10}{\sqrt{(-2)^2+6^2}} = \frac{20}{\sqrt{40}} = 3{,}16..,$$

While the distance from p3 (5,1) to p1-p2 is:

$$\frac{-2*5+6*1-10}{\sqrt{40}} = \frac{-14}{6{,}32} = -2{,}21..$$

The Algorithm for the Convex Hull

After making another observation, we can formulate the algorithm for the convex hull. Observation: The point that has the longest (largest) negative distance from a line pi-pj is itself a point on the convex hull (see fig4, point P and line I - A).

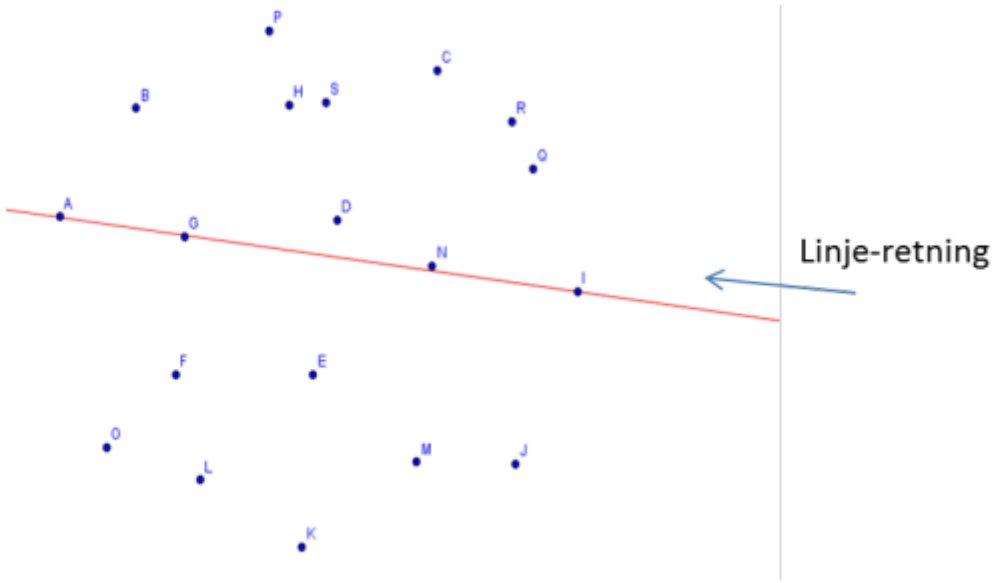


Fig.4. Starten på å finne innhyllinga fra maxx (I) til minx (A).

Fig4: The start to finding the convex hull from max_x (I) to min_x (A). **("line-retning" translates to "direction of line"**

Convex Hull Algorithm:

1. Draw a line between the two points we know are on the hull from max_x to min_x (I to A in fig4)
2. Find the point with the largest negative distance (or 0) from the line (in fig4 it's P). If multiple points have the same distance, just choose one arbitrarily.
3. Draw a line from the two points on the line to this new point on the hull (in fig5: I-P and P-A)
4. Continue recursively from the two new lines and for each of them find a new point on the hull with the largest non-positive distance (less than or equal to 0)
5. Repeat step 3 and 4 until there are no more points on the outside of the lines
6. Repeat steps 2-4 for the line min_x-max_x, and find all points on the hull under it.

This process is illustrated in fig5. You can see the process of finding the points on the hull above the line I-A. Expect to find about 1.4 * sqrt(n) elements in the convex hull from n randomly chosen points on the plane, although all values from 3 to n are possible (we get n, if all points are on a circle or a triangle or a rectangle or any convex polygon).

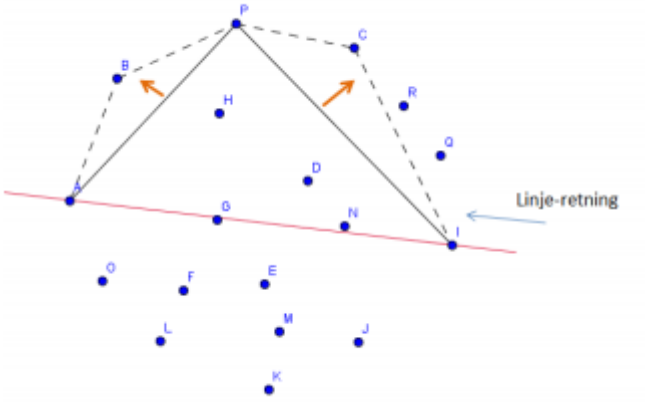


Fig5: We recursively find the point P on the hull from the line I-A (that with the largest negative distance from max_x-min_x), then C from P-I, R from I-C and lastly Q from I-R.

First, you are supposed to program this recursive algorithm sequentially, and then make a parallel version and measure speedups for n=100, 1000, 10 000, …, 10 million.

On generating n different points

a) The data structure for your n points should be two int arrays x and y, each of them with length = n.

```
int []x = new int[n], y = new int[n];
```

In order to fill these arrays with x and y values that does not create equal points, you will use the pre-written class NPunkter which you will find in the Oblig5 file.

b) For each value of n in your program, you should first create an object of the class NPunkter **(NPunkter translates to NPoints)**:

```
NPunkter p = new NPunkter(n);
```

and then fill x[] and y[] with a call to the method in the object p:

```
p.fyllArrayer(int [] x, int [] y);
```

(“fyllArrayer” translates to “fillArrays)

The reason why you must use this class, which you must not alter, is that all the deliveries for a given value of n should have the same points. Then it will be much easier for the correctors to correct the obligs (additionally, it is not easy to relatively quickly generate up to 10 million random points that you are certain are not equal, but that task is not a part of Oblig5).

c) The process of generating n points should be excluded from the reported execution times of the algorithm.

Efficiency

a) We are not really interested in the actual distance from a point to a line, it is just used as a relative measure to see which point “wins”, i.e., is furthest away from the line. The formula for the distance d can as such be simplified by avoiding the division.
b) It will be time-consuming if you have to run through the entire set of points P all the time to find the few points that are on the outside of a line. That is why you should let your program test on smaller and smaller sets of points
c) Remember to be critical regarding using threads instead of recursion (think “layers”, like quicksort). Threads should only be used at the top of the recursive tree.
d) Is `ArrayList<Integer>` fast enough to hold large sets of points, or should you use the class IntList with the “same” methods you need, and where the integers you are supposed to store resides in a simple integer-array?

The Assignment

In Oblig 5 you must:

- Program a sequential version of the algorithm described above.
- Parallelize the sequential implementation preferably using one of the two methods that will be described in the course lectures: either method 1 or 2.
- Use the precode *NPunkter17.java* and *Oblig5Precode.java* published separately. You must use the `drawGraph()` to draw the graph.
- It must be possible to specify the seed for the precode on the command line when starting the program.
- The program should be able to print the resulting convex hull to a file using the `writeHullPoints()` in the precode for n < 10000.
- Write a report on your findings. Report recommendations are on the course web.
- IN4330 students must in addition: run the Java Benchmarking Harness multiple times, and comment on what improvements you make to your code, and what the harness shows as empiric evidence of improvement.

Requirements

Oblig 4 Devilry deadline: May 3rd, 2023 23:59:00. The delivery should consist of:

a) The code for both the sequential and parallel solution, wrapped in a .zip file.
b) An output of the convex hull for n = 1000.
c) A report with both a table and a curve showing speedup as a function of n (= 100, 1000, … 10 million) and your assessment as to why your runtimes are as low as they are, and why you get the speedups you get. NOTE: runtime for n = 100 milion should be less than 14s and n = 10 million should be less than 2s. Include specifics of the computer you ran your tests on (modell, clock frequency, number of cores) and the size of the main memory, and if possible the size of the caches and delay (run the program latency.exe if you are on a Windows computer) and gladly a picture created by TegnUt when n=100.

Oblig 5 report recommendations have been published on the course page.

Tips

The following points are not requirements but describe areas where you might run into some trouble.

1) **The Sequential solution**
    a) **How to represent a point**
    Use the index in x[] and y[] - the contents of those will not change during runtime; they are only read.
    b) **Debugging**
    Very few, if any, manage to get their code right on the first run. We need to debug the code and that might be difficult on a graphical problem like this one. You can for instance use TegnUt to draw your lines and use a small number for n while debugging (n < 200). The call to TegnUt is made as such:

```
TegnUt tu = new TegnUt (this, koHyll);
```

    The first line is to draw the points and the convex hull that is expected to be added in to an `IntList coHull`.

    c) **Find points on the convex hull in the correct order**
    Tip: you are using two methods for the sequential solution (assuming here that you are using ArrayList, but this can trivially be changed by swapping it with a faster IntList):

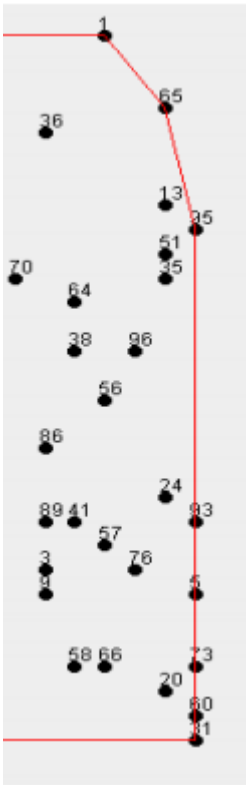
    seqMethod() which finds min_x, max_x, and starts the recursion. If you start on max_x-min_x, add max_x to the list of hull points (but not min_x yet), and start your recursion with two calls to the recursive method:

    seqRec(int p2, int p2, int p3, ArrayList m, ArrayList koHyll) which receives a set of points m (containing all points above or below the line p1-p2) and p3 that has already been found as the point with the largest negative or positive distance from p1-p2. koHyll is the set to which you should add the points from the convex hull in the correct order.

    Notice that you first “do a recursion” on the right side (line I-P in fig5) and then on the left side (line P-A) because we want the points to be added in a counter-clockwise order.

    You can let each call to seqRec add one point: p3 to the list of points on the hull, but where in your code should you do so? When do you add min_x to the list?

d) **Include all points on the hull in cases where multiple points are on the same line (distance = 0), such as on the right side in fig6, and in correct order.**



Remember that if you find that the largest negative distance == 0, there is no need to include p1 or p2 as possible new points (they have already been found).
Say that you have found p1=35 and p2=5 and you want to find all the points on the line between p1 and p2 (60 and 73) and you then have to test if a new point p3 has both x and y coordinates corresponding to those of p1 and p2. Then you can find one of the points with a call to seqRec above the line p1-p2 (31-5). Repeat recursively until there no longer are any points between the new p1 and p2.
The other points on the line (for instance, 5-95) will be found recursively similarly to 60 and 73.

2) The parallel solution

a) You could consider dividing the parallel case in two methods. The first parallel call from the main-thread (between the two lines where you time the parallel implementation). It will then find out how far down the call tree we should use threads. If we have k cores on our computer, it might be reasonable to move down to a layer in the call tree where there are approximately k nodes horizontally in the tree. Example: if we have 8 cores, a call the top of the tree level 1, then level 4 will have 8 nodes. Up until this level new threads will replace recursive calls, and from that level regular recursive calls can be used from each of the threads we started.

b) The first parallel method called from and being run from the main-method creates two threads. The following threads are born from these two threads and their offspring-threads. These are started in the parallel method, which is called from the run()-methods.

c) The other parallel method has the same parameters as the sequential one: p1, p2, p3, and the set of points to look within in addition to two more: the level of the tree and the set (IntList) of points on the hull that the thread has found.

d) The other parallel, recursive method decides whether to keep on creating threads or if we are going to start using recursive calls. Prior to that, we have found the point that has the largest negative distance and the set of points for the next call and we have also declared and generated to sets; one for each of those to recursions that are supposed to hold the convex hull that particular call finds.

e) This method strongly resembles the sequential method, and after we have reached the level where we seize to make new threads, we can actually just use the sequential recursive method.

f) Remember that when we start threads we have to first create both of them, and then wait (for instance by using join()) for them. If we do not, we do not get paralyzation.

Finding the points of the convex hull in the correct order (parallel solution)

Another problem you will face is adding points in the correct order. While it is pretty clear when to add points to the convex hull in the sequential solution, it is not as predictable in the parallel solution. A way to solve this is to use:

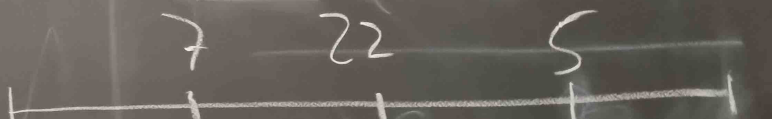
1. Threads started at the top of the recursion tree only traverse down to a certain level for all nodes of the tree (the level depends on the number of cores at our disposal)..
2. After we can no longer start new threads, each thread moves on to the sequential solution (on its side of data)
3. From each of these sequential executions we will get the points in the correct order (counter-clockwise)
4. The problem then is to join these sets of points that are individually in the correct order. Do this: add first the points the right-hand thread has produced, then the point found by the node itself, and then the points from the left-hand thread.
5. Point 4 is then executed (upon return after finding its points) upwards until we reach the top and is done.

EVAL

LEARN

HARD

$\sum (x - \bar{x})^2$



6 30 0 0

BUSY WAITING

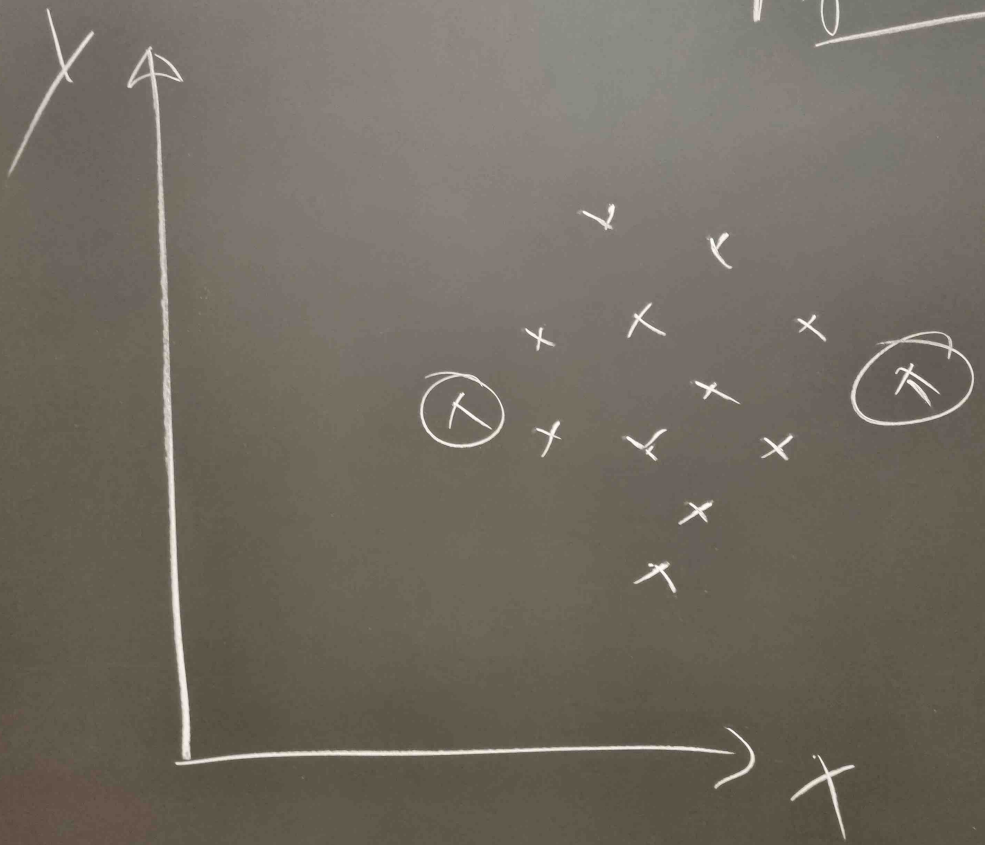
TIMED BUSY WAIT

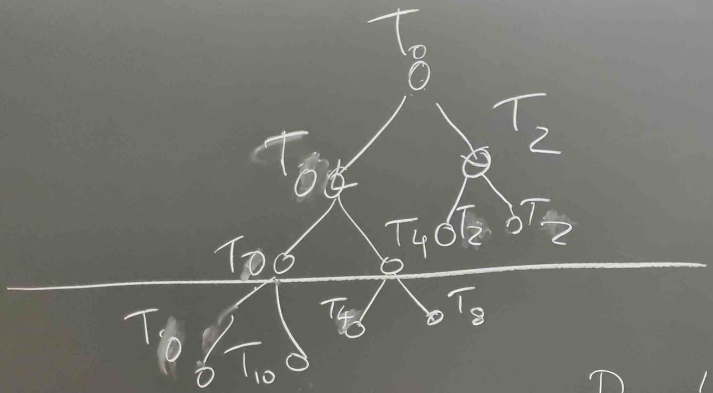
QUEUEING

SIGNAL ALL

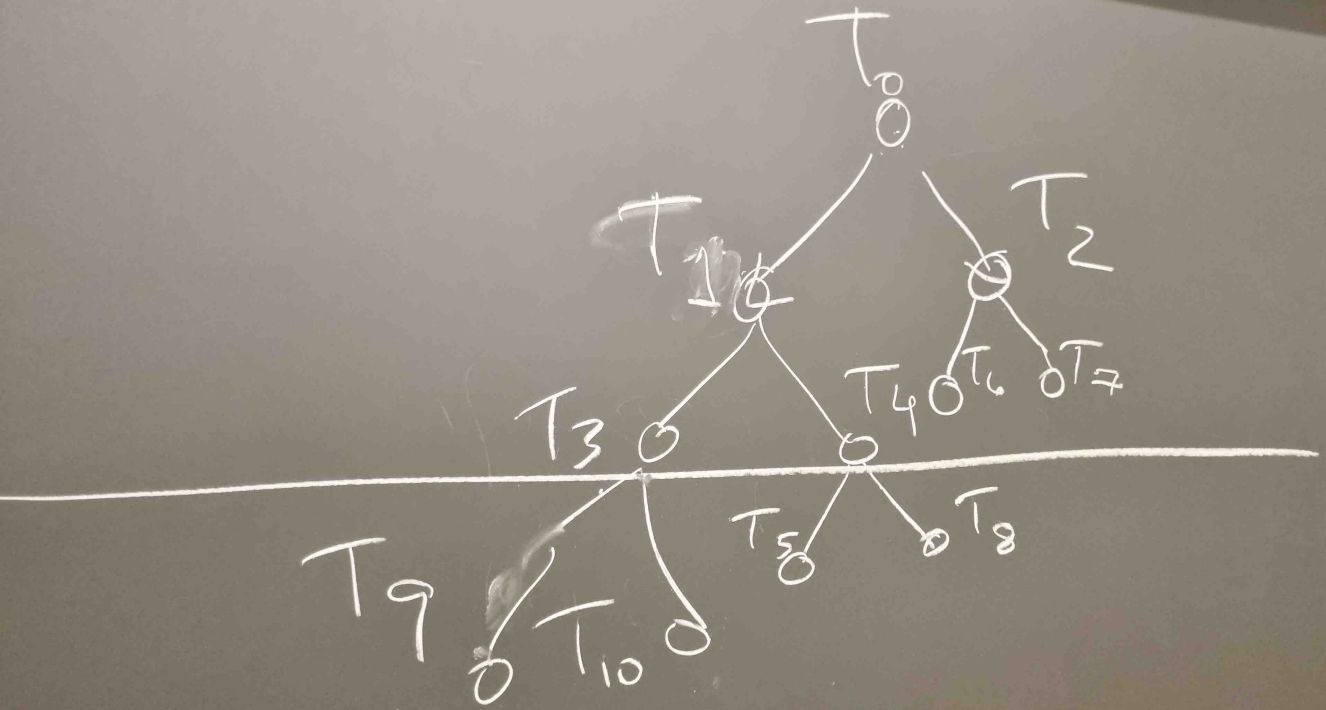
SIGNAL ONE

Agenda :





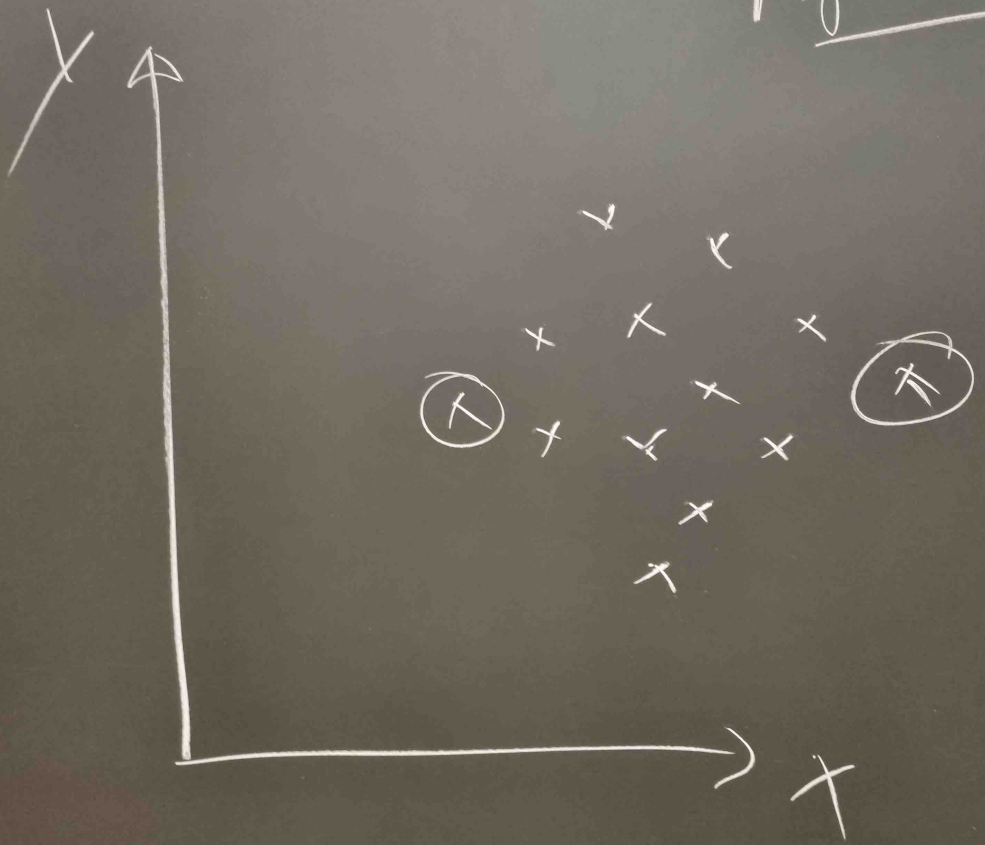
- Don't create a thread for sets with less than k points


T0
T1
T2
T3
T4
T5
T6
T7
T8
T9
T10

Agenda:

- lecture recordings ✓
- Coming lectures ✓
- Obl ig 4
- Parallelization of recursive algo

Agenda :



Effektiv Parallellprogrammering, kompendium i IN3030/4330

av

ARNE MAUS

PT (Programmeringsteknologi),
Institutt for informatikk,
Universitetet i Oslo

7. mai. 2023,
ver. 4

(Ris og ros, feil, forslag til forbedringer/utvidelser/strykninger og andre kommentarer sendes: arnem@ifi.uio.no)

Innledning

Dette er en første oppdatering av et kompendium i parallellprogrammering i Java på en flerkjerne maskin med felles lager. Et vesentlig tillegg er et nytt kap. 7 om lambda-uttrykk og strømmer og særlig da parallelle strømmer.

Dette stoffet foreleses på Institutt for informatikk i kursene IN3030/4030 på bachelor- og master-nivå. Det poengteres at dette kompendiet er tillegg til kurset som kan hjelpe til å forstå hvorfor dagens datamaskiner har stadig flere kjerner, hvilke mekanismer disse inneholder som gjør at programmer kan gå fortere alt etter hvordan vi skriver våre parallelliserte algoritmer.

Studentene har som forutsetning hatt det første kurset i Algoritmer og Datastrukturer og har hatt to introduksjonskurs i OO (objektorientert) programmering hvor de også har sett noen få eksempler på parallelle programmer i Java men ingen inngående opplæring i parallellisering. Kurset IN3030 starter derfor på bunnen med hva tråder er, men kommer ganske fort opp på et rimelig avansert nivå i parallellisering av algoritmer. Dette kompendiet er tenkt som støttenotat og utdyping av forelesningene og forelesningsfoilene i kurset.

Vi programmerer i dette kurset typisk for en vanlig PC /mobiltelefon med 2-16 kjerner, men programmene i denne boka vil også virke effektivt på mer uvanlige prosessorer som Intel Xeon Phi med 62 kjerner eller store server-prosessorer med 32-64 kjerner med felles hukommelse (som Graviton3 brikken med 64 kjerner laget med 55 milliarder transistorer på én brikke) . Det som her foreleses her er ideen om en datamaskin med felles hukommelse for flere prosessorkjerner, og at det er flere nivåer, typisk 3, med cache-hukommelse på hver kjerne (delt eller ikke delt cache er ikke viktig). Cache hukommelse forklares senere. Valg av programmeringsspråk, Java, er nok viktig for eksemplene i dette kompendiet, men det er få kjente grunner til at tilsvarende og nær identiske programmer ikke skulle være like effektive i Scala, C++, C#, Object C eller andre objektorienterte språk med tråder og med et felles adresserom (i hovedhukommelsen og cachene).

Det er to krav vi stiller til de parallelle programmene vi skal lage:

- De skal være **riktige** – produsere samme resultater som en riktig og rask sekvensiell løsning
- De skal være **mer effektive, klart raskere** enn en sekvensiell løsning av samme problemet

Viktig i dette kurset er hvordan man parallelliserer et riktig, sekvensielt program + og lage egne parallelle algoritmer. I dette kurset begrenser *vi oss til at* ***våre parallelle programmer skal gå på én PC med ett felles lager.*** Vi vil altså ikke ta opp bruk to andre aktuelle typer av maskiner, som også nyttes til å løse parallelle problemer som bruk av grafikk-kort og klynger av PC-er i nett.

Grunnen til at vi ikke omhandler bruk av grafikk-kort med mange tusen enkle prosessorer er at programmering av disse krever en helt spesiell kode og at de løser bare visse typer av problemer svært effektivt (mye data, med samme instruksjon utført på alle disse data i parallell)

Dagens virkelig store dataanlegg består klynger av flere millioner PCer koblet sammen i et raskt nett. Alle de metodene som læres i IN3030/4030 er aktuelle der, er det her mange ekstra problemer man får med en slik komplisert maskin – f.eks. strømforbruket (verdens største slik klynge i 2022 bruker ca. 30 000 KW), med tilsvarende store luft-kjølingssystemer. For slike klynger trenger vi også teknikker og programvare til det å spre ut en beregning ut på så mange enkelt-maskiner over et nettverk med den relativt store forsinkelsen i datanettet mellom maskinene (ca. 1000 ganger så stor forsinkelse sammenlignet med tidene til lesing og skriving i én PCs sin hovedhukommelse), og særlig det til sist å få samlet så mange delberegninger til ett, felles svar krever egne programmerings systemer og filsystemer.

Dere skal altså lære de mange problemene vi støter på og hvordan disse kan relativt enkelt kan løses med parallellisering av et sekvensielt program på én multikjerne PC. Kurset er *empirisk* (med tidsmålinger), og ikke basert på en teoretisk modell av parallelle beregninger. Vi oppfatter programmet som en god nok modell av det problemet vi skal løse. Vi trenger ingen modell av modellen. Siden slike teoretiske modeller, særlig ulike varianter av PRAM-modellen (Parallel Random Access Machine) brukes i mange lærebøker i parallelprogrammering, vil vi i dette kurset og i dette kompendiet kommentere hvor feilaktige anslag disse teoretiske modellene er, og gjør at både hvordan vi bør parallellisere våre algoritmer og hvilke kjøretider de faktisk da kan bli svært misvisende.

1. OM VESENTLIGE FORHOLD SOM PÅVIRKER EKSEKVERINGSTIDENE

Når vi kjører våre programmer vet vi at vi skriver det i et programmeringsspråk, i vårt tilfelle Java, som så kjører på selve maskinen med sine maskininstruksjoner og oppbygging med ulike typer elektronikk.

I det følgende vil vi gå gjennom de (forbausende mange) mekanismer, både de som har lager elektronikken og Java har lagt inn som vil redusere eksekveringstidene i stor grad, og som vi kan benytte oss av når vi skriver programmer. Jeg presiserer at det er ikke hele datamaskinen eller hele Java som beskrives – bare de mekanismene som vi som programmerere kan bruke til å skrive raskere programmer - både sekvensielle og særlig for oss – parallelle programmer.

Vi begynner først med datamaskinen før vi tar for oss Java, som nå er under stor utvikling med stadig nye begreper inkludert. I den grad det vil endre særlig hastigheten av våre programmer, vil det bli beskrevet. .

1.1 DATAMASKINENE

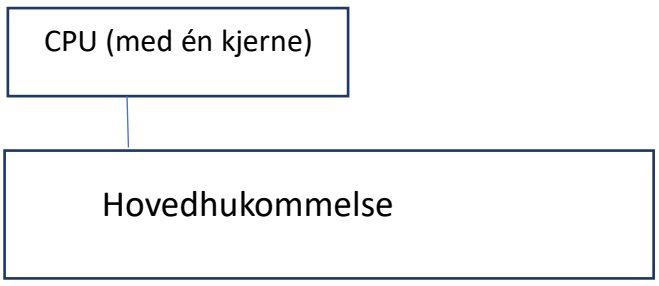
Hukommelses-systemet

a) **Lageret er byte-adressert**
Lageret i alle Intel, AMD og Apple/Arm -maskiner er byte-adressert – med 8 bit i en slik byte. Det betyr at den minste data-enheten vi kan lese og skrive fra CPUen til hukommelsen er én byte lang– f.eks er en **`byte`** variabel i Java er 8 bit lang (eks: **`byte a;`**), mens en vanlig **`int`** er 4 byter. Dette er kanskje ikke så viktig å vite i sekvensielle programmer, mens i parallelle programmer er det avgjørende at to parallelle tråder ikke samtidig prøver å skrive på samme byte i lageret, Da er det viktig å vite at hvis to tråder samtidig ønsker å skrive eller endre på ett av bit-ene i en slik byte samtidig, går det galt selv om det er ulike bit vi ønsker å endre på i denne byten.

b) **Cache-systemet.**
Når vi regner med data i programmet vårt, f.eks. skal utføre setningen: **`a = b + c`**, tenker vi at alle data er i en stor hukommelse og at variablene a, b og c der har hver sin plass i den. Vi leser/kopierer først verdiene av b og c fra hukommelsen, legger disse sammen og skriver resultatet ned i a-plassen . Det som skjer, er imidlertid langt mer komplisert.

1.2 LITT OM MASKINENS KONSTRUKSJON, CACHE OG MULTIKJERNE

Før 1980 var datamaskiner relativt enkle slik det framstilles i fig 1.1. Man hadde en beregningsenhet kalt CPU (Central Prosessing Unit). Den utførte én instruksjon av gangen i den rekkefølge de var spesifisert i programmet og leste, og skrev sine data direkte i hovedhukommelsen.



Figur 1.1 *Skisse av en datamaskin i ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variable) direkte i hovedhukommelsen.*

Fra 1980-tallet begynte imidlertid CPUene å gå mye raskere enn hovedhukommelsen. Ordbruken skiftet også, slik at nå snakker vi om en prosessor istedenfor CPU. Dagens avanserte prosessorer bruker om lag 150-200 ganger så lang tid å skrive til, eller lese fra, hovedhukommelsen som den tid det tar å utføre en enkel instruksjon (som å legge sammen to heltall). Her ville det ha blitt mye dødtid.

Svaret var å legge først én, senere flere mellomhukommelser (cache-hukommelser) mellom prosessoren og hovedlageret Nå er det vanlig med tre lag av cache-hukommelser som er hurtigere, men dyrere enn hovedhukommelsen. Når prosessoren 'tror' at den lagrer verdien av en variabel i hovedhukommelsen, lagres det først bare i nivå1-cachen (kalt L1, se fig 1.2), og prosessoren kan fortsette. Så snart som mulig lagres så disse data deretter i nivå2-cachen (L2), så i nivå3-cachen(L3) – og til sist i hovedhukommelsen omlag 200 klokkeenheter senere (1 klokkeenhet= ca. 1/3 milliardedels (1/3 nano-sekund) . Tilsvarende gjelder for lesning. Hvis prosessoren ikke finner de opplysningene den vil ha i noen av cachene, må det leses fra hovedhukommelsen og inn i alle cachene før prosessoren får adgang til data.

Flere kjerner. En annen viktig utvikling av prosessorene var at man etter ca. år 2005 ikke greier å få én prosessor til å gå fortere. Prøver man med en raskere klokke, vil prosessoren rett og slett først feile og så evt. smelte. Imidlertid greier man stadig å lage hver prosessor mindre og mindre ved at de transistorene den består av blir laget mindre. Hva skulle så databrikke-designere som Arm, Intel og AMD gjøre? De la flere prosessorer, heretter kalt prosessorkjerner eller bare kjerner på hver brikke. Vi fikk da maskiner med to prosessor-kjerner (dual-core), så med fire kjerner, osv. Det er uklart hvor dette ender, men det er i alle fall laget forsøksproduksjon av brikker med ca. 100 slike prosessorkjerner, og det er helt sikkert at utviklingen stopper ikke med det. I tillegg finnes maskiner hvor man har satt 2 eller 4 slike multi-kjerne prosessorer på samme hovedhukommelse. Dette er ikke uvanlig eller spesielt dyrt. I tillegg kan noen av disse kjernene kjøre 2 tråder hver i parallell; såkalt hyperthreading, fordi en del av elektronikken er duplisert i hver kjerne. Eksemplene i dette kapittelet er eksemplene fra 2017 testet på to slike maskiner, en med 8 tråder (=1 prosessor med 4 kjerner som hver har hyperthreading) og en med 64 kjerner (=4 prosessorer med 8 kjerner som hver har hyperthreading), mens 2022 og 2023 resultatene er testet på en nyere maskin med 8-tråder (4 kjerner). .

Det er også vanlig slik at hver kjerne har sin egen L1 og L2 cache, men deler som oftest L3 cachen med alle kjernene på samme brikke, men ikke med de andre prosessorene som evt. er i maskinen. Alle trådene i vårt program deler samme område i hovedhukommelsen. En viktig konklusjon på dette er at selv om en tråd som går på en av kjernene og har skrevet ned verdien av en variabel som alle trådene har utsyn til, så kan det ta lang tid før de andre trådene greier å se denne nye verdien fordi den f.eks. holder på å bli skrevet ned via alle cachene og det kan ta flere hundre klokkesykler før den oppdaterte verdien er i hovedlageret. Det er også ca. 10 til 30 registre per kjerne. Disse kan brukes til å holde de mest nyttede variablene i en beregning, slik som indeksen 'i' i en forløkke eller de mest sentrale variablene i en metode, som **`s`** i metoden **`sum`**

```
long sum (int [] arr) {
   long s = 0;
   for (int i = 0; i < arr.length; i++) {
         s = s + arr[i];
    }
    return s;
}// end sum
```

Prog 1.1 *Programeksempel, summering av verdiene i en array.*

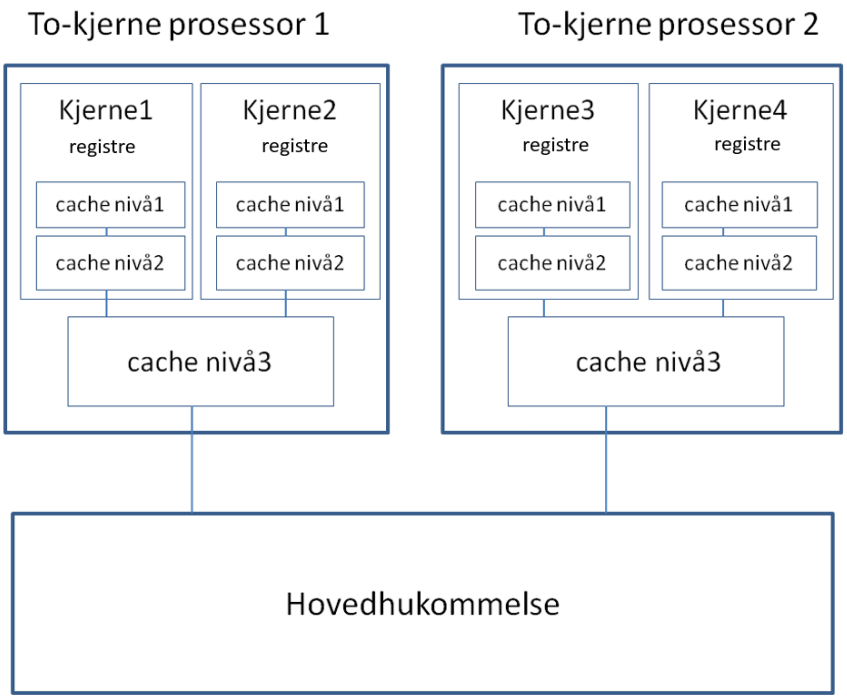
Det er Java-kompilatoren **`javac`** og senere kjøresystemet **`java`** som bestemmer hvor de ulike variablene i et program plasseres. Det er slik at alle variablene har sin plass i utgangspunktet i hovedhukommelsen. Men i koden fig. 1.1 ser vi at metoden sum inneholder to variable, **`s`** og **`i ,`** som ikke er synlige utenfor metoden og som ikke kan leses av resten av programmet etter at metoden er utført. Disse to variablene vil derfor i en optimalisering av koden bli lagt i hvert sitt register.

Parallelle instruksjoner I de siste årene har det kommet en rekke maskin-instruksjoner, som sammen med en rekke nye registre i kjernen, kan de f.eks både med en og samme instruksjon multiplisere sammen to og to av disse registrene og summere resultatene fra disse 10-20 sett av registre. Dette er også en optimalisering av Java-koden kan nytte seg av.

```
void add1 (int tall, int ant) {
     for (int i = 0; i < ant; i++) {
          tall++;
     }
  }// end add1
```

Prog 1.2 *Metode som øker variablen* ***tall*** *ant ganger.*

I det programeksempelet vi har i 1.2, vil vi illustrere flere viktige poeng i parallellprogrammering. Først må vi vite at operasjonen tall++ ikke blir utført som én operasjon, men egentlig er tre operasjoner: Først les verdien av tall, så legg **1** til denne verdien, og til sist skrives den nye verdien ned i variabelen tall. Som vi vet betyr dette at både gamle og nye verdien går via L1 cachen og veien ned til hovedhukommelsen er lang. Vitsen med disse cachene er at selve beregningsenheten bare forholder seg til sin L1 cache, leser og skriver i den, mens resten av systemet stadig forsøker å holde de andre cachene og hovedhukommelsen oppdatert. Beregningsenheten greier altså å gå så fort som L1 cachen greier å lese og skrive data, nesten hundre ganger raskere enn hovedhukommelsen hvis den ikke må vente på data fra en av de langsommere cachene eller hovedhukommelsen.



Figur 1.2. *To dobbeltkjerne prosessorer i en maskin med ulike hukommelser. Når det går parallelle tråder på hver av disse kjernene, ser vi at de som oftest ser ulik verdi på en felles variabel (eks. int i) i hovedlageret, hvis noen av trådene leser på en slik variabel samtidig som en annen tråd endrer dens verdi ved skriving. Dette fordi de ulike cashene ikke hele tiden er fullt oppdatert på siste verdi som er skrevet. Samtidig lesing og skriving på en variabel av mer enn én tråd må derfor alltid synkroniseres.*

Man kan jo til slutt spørre hvorfor man har alle disse lagene med hukommelse. Siden man greier å lage rask L1 cache, hvorfor kunne ikke all hukommelse vært slik? Poenget er at slike cacher er laget på en mye dyrere måte enn hovedhukommelsen, hver byte i cachene tar større plass og krever mer strøm. Det er også slik at måten disse cachene adresseres på en måte som forutsetter at de er relativt 'små'. Det ville kort sagt ikke lønne seg eller mulig å lage all hukommelse slik.

Typiske størrelser og forsinkelser måles i hvor mange tikk (cycle) som klokka i prosessoren må gjøre før en operasjon er ferdig. En prosessor som er på 2GHz, har da en klokke som gjør 2 milliarder slike tikk per sekund. Forsinkelsene i de ulike delene av hukommelses-systemet kan da være:

Registre i kjernen	størrelse = 0,1-1 Kb,	tidsforsinkelse = 1 cycle
Cache nivå 1	størrelse = 32Kb	tidsforsinkelse = 2 cycle
Cache nivå 2	størrelse = 512Kb	tidsforsinkelse = 7 cycle
Cache nivå 3	størrelse = 4096Kb	tidsforsinkelse = 16 cycle
Hovedhukommelse	størrelse = 8 – 16 GB	tidsforsinkelse = 190 cycle

Ekstra L3 cache. En nyhet i 2021/2022 er at en produsent (AMD) ha laget sin siste prosessor for spill og tekniske beregninger så tynn at det er mulig å legge en ekstra L3 hukommelse rett oppå brikken slik at den faktisk da får ca. 90 Mbyte totalt med L3 cache. Vitsen med å gjøre den tynn, er at det fortsatt er mulig å kjøle brikken selv med den ekstra L3 hukommelsen lagt oppå den.

Cache-linje størrelsen Når cache-systemet leser data fra hovedhukommelsen opp til nivå 3, så nivå 2 og til sist nivå1 cache, er det ikke én byte ad gangen, men 64 byte hver gang (fordi det tar så lang tid, er det fordelaktig å laste opp mer enn det det er bedt om). Hvis programmet trenger før eller siden disse bytene, ser vi at programmet vil gå mye raskere enn om man skulle gå helt ned i hovedlageret hver gang programmet trengte neste heltall, flyttall eller neste byte.

Prefetch mekanismen Hvis programmet leser eller skriver tilfeldige adresser i en stor array med én indeks (større enn L3 cachen): a[i] , a[k] a[j], a[j],. .. så vil dette gå vesentlig langsommere enn sekvensiell aksess : a[i], a[i+1]. Spesielt oppstår denne effekten når vi bortover ***radene*** i en særlig i en litt større to-dimensjonal array (så stor at det ikke er plass til hele arrayen i L2 eller L3 cachen), og programmet leser f.eks. a[i,j], a[i,j+1], a[i,j+2] vil en mekanisme i prosessorkjernen starte å lese neste cache-linje på 64 byte ***før*** programmet har bedt om det selv. Dette gjør at programmet går vesentlig raskere – mindre venting på data fra hovedhukommelsen. Denne mekanismen virker like bra hvis vi leser radene i en array baklengs fra slutten mot begynnelsen.
Merk at dette gjelder ***radvis*** lesing av en array. Leses derimot arrayen **kolonne**-vis (a{i,j] a[i+1,j] , a[i+2,j], .. vil elektronikken måtte lese en ny cache-linje for hver aksess, og vi vil få ingen hjelp av at vi allerede har lest 64 byter i cache-linjen som inneholder a[i,j] og prefetch-mekanismen gjenkjenner ikke dette som sekvensiell lesing. Lesing av neste kolonne-element vil derfor medføre at hver gang foretas en ny lesing 'helt' fra hovedlageret med en forsinkelse på ca. 200 nano-sekunder mot lesning av L1 cachen på 2 nanosekunder for radvis lesing av hvert element. En slik 'bom' på hva vi skal lese neste gang kalles en cache-miss.

Pipeline: Ikke alle maskininstruksjoner tar like lang tid, og de er da delt opp i flere mikroinstruksjoner som samlet sett løser oppgaven som den større instruksjonen skal løse – f.eks divisjon. En kjerne vil da ved neste klokke-tikk prøve å starte første mikroinstruksjon i neste instruksjon, så igjen ved neste tikk neste instruksjon, ...osv. Kjernen kan holde da på med å beregne typisk inntil 5 maskin-

instruksjoner samtidig på forskjellige grader av fullføring. Dette kalles en 'pipeline. Vi vil senere se at f.eks. multiplikasjon tar mye lenger tid (2-7 ganger så lang tid) som addisjon.

Trådbytte: er når en tråd som utføres på en av kjernene, avbrytes og en annen tråd overtar kjernen og kjører sitt program på denne kjernen. Den tråden som holdt på med sine beregninger må stoppes og innehold av de registrene som denne avbrutte tråden hadde til sine beregninger må lagres i hukommelsen. Så kan register-innholdet til den nye tråden lastes opp fra der den andre tråden hadde lagt sitt registerinnhold. Først når registerinnholdet fra den nye tråden er i registrene, kan denne startes. Når vi har flere tråder enn vi har kjerner i en prosessor, og det har vi alltid når vi teller med de over 1000 tråder vi har i operativsystemet, så må de ulike trådene bytte om på å kjøre på kjernene. Selv om operativsystemets tråder i sum ikke bruker mer enn ca- 10% av prosessoren klokkesykler fordi de i all hovedsak venter på I/O eller andre begivenheter, så tar de noe tid. Også, hvis ingen tråd ønsker å kjøre på en prosessor, er det et enkelt lite program som kjøres: *idle-loopen*- det enkleste program vi kan tenke oss som bare går rundt i en evig løkke, og som med en gang vil foreta trådbytte med en tråd som har noe å gjøre. Når idle-løkken kjøres vil også prosessoren søke å redusere klokkehastigheten slik at den sparer strøm. Den PC-en som noen av eksemplene i dette kompendiet er kjørt på kan variere klokkehastigheten mellom 1.2 GHz og opp til 3.4GHz. Kort sgt, prosessoren med sine kjerner stopper aldri, og kjernene bytter ofte om på å kjøre de ulike trådene.

Spekulative beregninger Når koden som utføres inneholder en test (i f.eks. if- while eller for-løkke) inneholder kjernen ekstra registre og elektronikk slik at den starter med å regne på begge grenene av utkomst av testen – både om testen gir sann eller falsk. Når testen er ferdig beregnet vil kjernen beholde beregningene som fulgte etter med det riktige sanne svaret på testen og fortsette der. Beregningene som fulgte etter at testen feilet vil bli strøket.
En annen enklere løsning på dette er at når en løkke er ferdig, så stoppes alle andre instruksjoner i pipelinen . De andre instruksjonene må da starte om igjen, mens koden for den andre utfallet (at f.eks. løkken var ferdig) kan begynne å eksekvere.

IKKE så viktig her: Program-cache og cache for virtuell-tabellene mm. I tillegg til cache-system for data slikt det er beskrevet ovenfor er det også en tilsvarende rekke av cacher for den optimaliserte koden. Også den virtuelle adresseringsmekanismen har cache-områder for sine tabeller, dvs at alle programmer får byttet ut de øvre delen av adressefeltet i instruksjonene med den «virkelige» adressen til hvor data ligger i hovedhukommelsen. Dette er et system som gjør det lettere å skrive programmer ved at *de alle kan skrives som om at data ligger sammenhengende* i *hovedhukommelsen*. Tidligere var det slik at når lageret ble fullt ble større deler av data midlertidig ble skrevet til disk (nå til SSA- 'disken'). I våre dager har man funnet ut at visse dataområder helles kan komprimeres med ZIP-algoritmen i selve hurtighukommelsen, slik at plass blir gitt programmer med høyere prioritet (disse ZIP-komprimerte områdene kan senere dekomprimeres når det blir plass til dem og program etterspør disse dataene).

Alle mekanismer i dette avsnittet bør man vite om, men de er klart mest viktige for de som skal skrive operativsystemer eller kompilatorer o.l., Det er mindre viktige hvis man som programmerer av brukerprogrammer for å påvirke kjøre-hastigheten. Hyggelig er det jo at noe av kompleksiteten i elektronikken er allerede tatt hånd av andre når vi skal skrive våre algoritmer og lage større datasystemer.

Hvorfor er hukommelsen så 'langsom'. Når man kjøper en datamaskin, kan man se som om hovedhukommelsen har om lag samme klokkehastighet, f.eks, 2600 MHz som CPU-en, men hvorfor tar det da så lang tid å lese og skrive i den. En lesning av en cache-linje på 64 byte sendes over data-kanalen som en ordre med startadressen til hukommelsen over en linje som enten kan sende 64 bit i parallell, eller mye raskere serielt 1 bit av gangen. Deretter skal data leses i hukommelsen, Den er meget billig,

men også meget kompakt laget. Å lese data der tar ca. 15-17 klokkesykler før de er klare for å sendes ut på datakanalen til CPU-en. Siden vi ber om 8 slike forsendelser, skulle det ta ca. 8*16 cycler – dvs ca 125 cycler bare for lesingen i tillegg til overføringen. Det korte svaret er da at hovedhukommelsen er meget rimelig og kompakt organisert, men at det går ut over hastigheten.

2. JAVA .

Java er et språk som er under sterk utvikling. Oracle, som nå eier Java, lager hvert år flere nye versjoner. Det som i hovedsak kommer til er nye begreper og konstruksjoner som records (som er objekter bare med data og ikke metoder). Prinsippet er at hver tredje versjon som lages er stabil (Java 5, Java 8, Java 11, Java 14 og Java 17,…) og strømmer beskrevet i kap. 7. Disse versjonen vil bli vedlikeholdt (feilrettet og tilpasset stadig nye versjoner av operativsystemer) i mange år fremover mens de to versjonene mellom disse er 'forsøksversjoner' hvor nye konstruksjoner kan komme og gå eller bli endret. Powerpointfoilene er i kurset er hovedsak laget med Java 5 og Java 8, mens noen av hastighetsbetraktningene i dette kompendiet er laget med Java14 (komplett liste over java -versjoner på:https://www.java.com/releases/, som sier f.eks at neste stabile versjon 20 vil komme Juli 2023, med en tidlig.versjon i mars 2023). For å få bedre forklaringer på det nye som kommer i Java anbefales å abonnere på 'Java-magazine' (gratis) : https://blogs.oracle.com/javamagazine .

2.1 JAVA OPTIMALISERING, DEL 1

I det overstående er viktige mekanismer i elektronikken som kan øke hastigheten på programmet vårt, men dette er mekanismer som i hovedsak operer på maskinkodenivå, og vi skriver jo programmene våre i Java. Det er imidlertid mye vi kan oppnå med hvordan vi skriver Java-koden:

1. Data som vi opererer på bør være minst mulig slik at de data programmet beregner på til enhver tid om mulig passer inn i L1 eller L2 cachene.
2. Uansett, prøv å lese og skriv data mest mulig sekvensielt for å utnytte prefetch mekanismen, og spesielt må vi i to-dimensjonale matriser lese/skrive data langs med radene, aldri langs kolonnene (i Java og nesten alle andre programmeringsspråk). Fortran er derimot annerledes, og lagrer data i to-dimensjonale data kolonnevis, og da må vi i Fortran lese/skrive disse matrisene kolonnevis).
3. Lag gjerne mange små metoder som hver løser ett enkelt problem (som f.eks. summen av elementene i en array, eller som finner neste primtall i en primtalls-array). Den mekanismen vi beskriver nedenfor som optimaliserer koden kan ikke like lett optimalisere lange metoder med mange løkker som mange små metoder som kaller hverandre.

Vi har i tidligere kurs lært at java-kompilatoren **`javac`** oversetter vårt Java program til en enkel og kompakt kode, bytekode, som kan sees på som instruksjonene til en byte-maskin som utfører disse byte-instruksjonene; noe tilsvarende som Python utføres idag. I den første varianten av Java, Java1 var det som skjedde – man ga bytekoden til **`java`** som så leste de ulike bytekodene og så utførte dem en-etter-en.
Det som nå fra og med Java 6 skjer er følgende:

1. Første gang et objekt fra en klasse genereres eller en metode kalles i koden, blir den først oversatt til maskinkode på den maskinen man kjører på (just-in time kompilering). Og det er denne maskinkoden som utfører programmet vårt. Det er klart raskere enn å tolke byte-koden som instruksjoner til en byte-maskin. Merk at det bare er de delene som utføres som blir oversatt til maskinkode – resten forblir i byte-kode.
2. Utføres en metode flere ganger (si 10-100 ganger), så optimaliseres den oversatte maskinkoden, Instruksjoner kan bli byttet om og forenklet, men den 'optimaliserte' koden gir samme svar som en ikke-optimalisert kode. Denne koden er nå mye raskere enn den ikke-optimaliserte maskinkoden

3. Utføres en slik optimalisert metode mange ganger – si mer enn 20 000 ganger som blir koden ytterligere optimalisert, og går da enda mye raskere
4. Kalles en metode enda flere ganger. f.eks. over 100 000 ganger, blir koden for denne metoden ytterligere optimalisert (nå for 3dje gang) og kan igjen gå enda raskere.

Denne optimaliseringene som gjør at noen operasjoner i Java kan gå fra 4 - 100 000 ganger fortere, er beskrevet i tabellen nedenfor hvor ulike java-elementer og to sorteringsalgoritmer testes.

Tider i usek per iterasjon som funksjon av n, antall iterasjoner											
n	1	2	3	10	100	1000	10000	100000	1000000	X bedre (from n=1)	X bedre (from n=2)
for-loop, len = 100	0,4	0,2	0,1	0,022	0,020	0,015	0,002	0,002	0,00000	**181**	**90**
metodekall	4,5	0,9	0,3	0,098	0,087	0,063	0,005	0,005	0,00043	**10465**	**180**
int[] new , , len = 100	1,1	0,3	0,2	0,117	0,108	0,275	0,160	0,160	0,09815	**11**	**2**
array kopi for-løkke , len = 100	2,4	1,7	3,2	1,980	1,880	0,889	0,018	0,237	0,01217	**197**	**7**
System.arraycopy , len = 100	4,4	0,6	0,3	0,124	0,120	0,043	0,025	0,022	0,02008	**219**	**27**
new Thread,start&join	1164	362,9	224,0	197,424	174,278	172,948	175,820			**7**	**2**
new Class C m.metodekall	791,4	15,3	1,8	0,268	0,220	0,045	0,002	0,004	0,00274	**288832**	**3438**
int [] a skriv , len = 100	0,8	0,7	0,1	0,036	0,035	0,027	0,023	0,008	0,00659	**121**	**93**
int [] les ,, len = 100	0,3	0,3	0,0	0,028	0,026	0,022	0,015	0,006	0,00638	**47**	**54**
double to long	3,7	1,2	0,3	0,218	0,164	0,066	0,042	0,000	0,00004	**92500**	**60000**
insertSort double[] , len = 100	93,1	164,0	150,1	55,343	7,266	1,754	1,634	1,618	1,61902	**58**	**101**
Arrays.sort double[] , len = 100	404,4	73,1	60,9	56,506	6,028	1,357	1,165	1,116	1,11126	**364**	**65**

Figur 2.1 *Kjøretider per kjøring i* ***2022*** *som funksjon av antall ganger eksekvert i µsekunder (million dels) for ulike Java-elementer og for to sorteringsprogrammer: Ett brukerskrevet med kode i testprogrammet (insertsortering av flyt-tall) og et fra Java-biblioteket (Quick-sortering av flyt-tall). De to siste kolonnene viser speedup, hvor mange ganger fortere en eksekvering er etter 1 million kjøringer, regnet ut fra tidene for første kjøring (n=1) eller andre kjøring (n=2). Testene er kjørt på en AMD Ryzen 5 3500U PC med Java versjon 14.01 i Windows 10.*

Kommentarer til tabellen:

1. Vi ser at optimaliseringen er meget sterk for nesten alle konstruksjoner, men særlig metode-kall og det å lage et objekt av en klasse (samt konvertering av flyt-tall (double) til heltall) effektiviseres vesentlig. Dette er viktig for vår programmering – å dele opp vårt program i klasser med mange mindre metoder, koster lite eller ingen tid når programmet har kjørt 3-10 ganger eller mer.
2. Vi ser at de to kolonnene til høyre regner ut Speedup (SU) henholdsvis fra tiden det tar å kjøre første gang og andre gang. Grunnen til også å regne ut SU fra andre kjøring er at da er all tisdsbruk som kompilering til maskinkode og henting av klassen evt. fra Java-biblioteket unnagjort. Vi ser dette spesielt i eksempelet Arrays.sort som første gamg tar 404 µs mot innstikksort med 93 µs som har koden liggende i testprogrammet. Derimot er optimelisering av maskinkoden der den senere foretas, inkludert i kjøretidene (som f.eks. Inntikksort andre og tredje gang).
3. Det er meget tilfredsstillende at egen skrevet brukerkode som innstikkSort kan optimaliseres opp mot en faktor 60-100.
4. Av det overstående kan det se ut som f.eks. Innstiksort blir effektivisert for all mulig bruk i programmet. Det er galt. Optimalisering av metodene består bl.a. i at hele koden for Innstikksort (og alle andre metoder) blir stappet inn koden der den *kalles fra* - erstatter kallet med selve koden. Det betyr at hvis vi bruker og kaller Innstikksortering fra et annet sted i programkoden må den gå igjennom samme optimalisering. Dette gjelder også optimalisering av å lage et objekt av en klasse (**new)**

2.2 PROSESSER OG TRÅDER

Et program (en prosess) i Java består nå av en eller flere tråder. Hver av trådene er ett sekvensielt program som deler i tillegg til at de kan ha hvert sitt private del av hukommelsen, og koden i trådene utføres ovenfra og nedover slik vi tidligere har lært at programmer oppfører seg. Har vi flere tråder i programmet vårt har vi et parallelt program.

Vi vil senere kommentere på nye typer av tråder og enklere begreper som kan bli implementert i Java.

1. Når man starter java-programmet, får vi én tråd: maintråden. Den tråden starter med å utføre metoden main(). Fra main-tråden kan så programmet vårt starte flere andre tråder som vi har skrevet kode for.
2. For å få disse andre trådene må de programmeres som en egen klasse som er en subklasse av klassen Thread – eller de kan være en klasse som implementerer grensesnittet Runable. Alle de objektene vi så lager av denne subklassen vil da inneholde en egen tråd som vil utføres i parallell med de andre trådene i vårt program.
3. Denne tråd-klassen, her kalt Para, legges helst inni den klassen som inneholder main() – metoden, og skal selv inneholde en metode public void run() { ..] som dere må skrive selv. Denne metoden run() tilsvarer på mange måter i tråd-objektet main() metoden i hovedprogrammet, og er den metoden som kalles av systemet når tråden er laget og startet:

```
Thread[] t = new Thread [antTraader ];
for (int i = 0; i< c; i++) {
    t[i] = new Thread(new Para(i));
    t[i].start();
}
```

4. For senere i programmet å vente på alle disse `antTraader` stk. trådene blir ferdige, kan man utføre flg. setninger:

```
for (int i = 0; i< antTraader; i++){
    try{t[i].join();} catch (Exception e) {};
}
```

   Kurset inneholder fler andre måter, f.eks. vente på en egen CyclicBarrier i maintråden på at alle de trådene man har startet er ferdige.

5. Legg merke til at det nye trådobjektet er en parameter til klassen Thread.
6. Man kaller ikke si `run()` direkte, men med å kalle metoden `start()` i dette nye trådobjektet, som gjør mye bak kulissene for å lage denne nye tråden og til sist kalles `run()` i trådobjektet.
7. Trådobjektene starter altså med å utføre metoden `run()` som du har skrevet – denne metoden tilsvarer da metoden *main()* for maintråden.
8. En tråd avsluttes når den har utført siste setning i sin main() eller run() – metode.
9. Ingen tråder må drepe/avslutte en annen tråd, men ofte vil en tråd legge seg å vente på at en bestemt operasjon (utført av andre tråder) er ferdig.
10. Ikke før alle trådene, også main-tråden i et program er avsluttet, er programmet ferdig.
11. Alle disse trådene deler samme adresserommet i hovedlageret – de ser de samme variablene (data), classer og metoder som er deklarert der ut fra sitt skop (sitt utsyn til deklarasjoner).

12. Trådene utføres også samtidig og på hver sin kjerne i CPU-en hvis vi har en kjerne for hver tråd. Er det flere tråder enn kjerner, vil operativsystemet prøve å la trådene dele på bruken av kjernene. Dette ordner operativsystemet (Windows, Linux eller MacOS).
13. Operativsystemet har i tillegg en rekke tråder for å ulike oppgaver (administrasjon, I/O, nettet, vinduer på skjermen,…,osv). De denne setningen ble skrevet hadde Win10 operativsystemet 3789 tråder som var aktive, men nesten alle disse trådene lå og ventet. I sum tok de mindre enn 10% av maskinens kapasitet. Disse systemtrådene står i motsetning til de trådene vi skal skrive i IN3030 som hver vil forsøke, grovt sett, å bruke *hele kapasiteten* til en kjerne.

Med flere tråder har vi da et parallelt program-system hvor flere sekvensielle programmer kjører samtidig. De fleste problemene med slike parallelle systemer er når to eller flere tråder vil skrive på samme variabel. Vi må da synkronisere trådene (mer om det senere), eller hver tråd må lokalt ha en kopi av slike data og skrive/lese på disse. Senere samstilles data fra disse lokale kopiene.

2.3 NYE KLASSER, SØPPELTØMMING MM

For å lete kodeskrivingen er det i Java 17 innført begrepet ‘record’ som gjør det lettere å behandle klasser som bare inneholder data (eks: Punkt med en x og en y-verdi), Videre er det rent statiske classer som inneholder data som ikke skal endres etter at de er laget og metodenavn som parametere. Viktigst er kanskje at det er definert tekstblokker som er et antall linjer med tekst. Dette er syntaks som gjør det enklere å skrive kode, men siden det implementeres som ‘vanlige’ klasser er det ikke begreper eller tillegg til Java som gjør at paralelle programmer går fortere .
Det som gjør at Java 17 programmer nok går fortere enn Java 8 programmer er at søppeltømmingsalgoritmer i Java 17(dvs de objektene som er laget under kjøring og som nå ikke lenger kan brukes fordi ingen av trådene har en peker til dem) nå er byttet helt ut med den ‘gamle’ algoritmen med en ny som er mer inrementell, Ikke tar alt søppel med en gang, men gradvis fjerner ‘søppel’.objekter.

Et annet prosjekt (Valhalla) prøver å lage en mer effektiv inn-utpakking av enkle variable , slik en ‘int’ blir pakket inn som en Integer med et objekt rundt seg, eksempelvis i ArrayList <Integer>, som klart tar lenger tid og plass. Dette kompendiet har pekt på dette problemet i kap 13, hvor en enkel int[] nå klart er raskere enn en ArrayList<Integer>. Arbeidet med å få basale typer (som byte, int og double) og den motsvarende (Byte, Integer og Double) inn som felles begreper med felles metoder vil bli først komme i Java 22 i 2023 eller 2024 (https://openjdk.org/projects/valhalla/). Poenget med Valhalla er å endre hvordan generiske typer som Integer er definert i Java slik at man kan blande basale typer og generiske typer og at generiske typer kan både få mer effektivitet og ikke mer plasskrevene enn basale typer som ‘int’.

3 MER OM PARALLELLE PROGRAMMER I JAVA

Vi kan altså få altså feil i programmene våre hvis mer enn én tråd samtidig skriver på en variabel som er felles. Vi ser at alle andre trådene som ønsker å skrive samme stedet da må stoppes og vente mens den første tråden blir ferdig med å skrive. Dette kaller vi å synkronisere trådene. I tillegg sørger for at alle tradene ser samme verdier i alle felles datastrukturer

Parallell programmering er vanskelig, og det er derfor utviklet flere synkroniseringsmåter og biblioteker for mer strukturert parallell programmering i Java. Vi skal her gjennomgå bruk av en særlig nyttig synkroniseringsmetodikk; Bruk av barriere-synkronisering, men starter med en gjennomgang om både hvorfor vi trenger å programmere mer parallelt og særlig hvorfor det så komplekst.. Etter denne gjennomgangen kan vi lett få inntrykk av at det er umulig å få parallelle programmer riktige. Det er ikke tilfellet, men for å få til det må man følge noen klare og enkle regler. Bryter man bare én av disse, vil det kunne gå veldig galt.

3.1 HVORFOR LAGE PARALLELLE PROGRAMMER MED TRÅDER?

Det er i hovedsak tre grunner til at vi kan ønske å ha parallellitet i et program:

1. Man skiller ut visse aktiviteter som går *langsommere* i en egen tråd, som tegning av grafikk på skjermen eller søk i en database. Resten av programmet kan da fortsette uten opphold.
2. Logikken i programmet er slik at det naturlig består av en rekke uavhengige aktiviteter som bare sjelden trenger å bruke felles data. Hvis hver slik aktivitet programmeres med hver sin tråd blir programmet faktisk ofte *lettere* å skrive. Et godt eksempel er at du lager et system for direkte salg av flybilletter (eller innlevering av oppgaver i et kurs i programmering). Siden mange samtidig skal kunne gjøre dette, skriver du programmet slik at en tråd snakker bare med én kunde, og betjener bare den. Det er relativt lett. Dersom flere 'kunder' melder seg, starter vi bare en ny slik tråd for hver ny kunde. Vi ser at av og til må disse ha adgang til felles data, som for eksempel de ledige plassene på en bestemt flyavgang, og da må vi synkronisere trådene og sørge for at bare én får tilgang til å endre felles data av gangen. Feil som kan oppstå da må vi håndtere, men jevnt over kan disse trådene operere i full parallell.
3. Vi ønsker i dette kurset å bruke parallelliteten til å få visse beregninger til å gå *raskere*! Eksempler kan være store ingeniørberegninger, generering av bilder i spillgrafikk eller som det eksempelet vi til sist skal se på, sortering av større datamengder.

Vi skal i det etterfølgende basere oss på oppgaver av type 3, at vi ønsker et raskere program, men mesteparten av det vi skriver kan også brukes direkte i de to andre tilfellene.

4. OM BRUK AV SYNKRONISERINGSPRIMITIVER OG LÅSER

Vi har sett at det er store problemer når ulike tråder vil skrive nye verdier inn i samme variabel – ulike tråder ser ulike verdier. Det er faktisk også vanskelig å avgjøre for én tråd når en annen parallell tråd er ferdig.
Til å løse disse problemene har man innført flere typer av låser i Java. En lås er en mekanisme som kan stoppe og la en eller flere tråd(er) vente. Tråden kaller på låsen og låsen gjør en test, og avgjør om kallende tråd må vente eller ikke. Felles for låsene i Java er følgende gode egenskap:

Når flere tråder gjør et kall på ***samme lås,*** vil alle disse trådene være sikret at all skriving trådene har gjort på felles variable ***før dette kallet*** er synlig for alle de andre *etter kallet*. De ser da samme verdier på felles variabler**.** Da blir bla. alle felles variabler for disse trådene som er endret skrevet ned i hovedhukommelsen fra cashene før trådene kan fortsette.
I Java er det mange titalls forskjellige låser som kan synkroniserer trådene. Vi skal i hovedsak i våre løsninger bare bruke tre av disse som er kortfattet beskrevet i neste avsnitt. Men for å illustrere visse problemer skal vi også i kurset nytte noen andre slike låser senere. Til alle slike låser/synkroniserings-mekanismer er det en rekke metoder (totalt 19 stk. for ReentrantLock) hvor en bruker kan spørre om hvor mange andre tråder som venter på denne låsen, spørsmål om å neste i køen
osv.
Vi vil illustrere dette senere med programmer som benytter tre typer av låser: CyclicBarrier som sikrer at vi vet når aller trådene er ferdige og som vil stoppe tråder og la dem vente inntil alle trådene er ferdige med en bestemt del av koden. En nyttig og den raskeste låsen er ReentrantLock for å beskytte en bestemt metode fra at flere tråder samtidig kan utføre den metoden. Bare en tråd slipper inn ad gangen og andre tråder som litt senere kaller denne metoden må vente til den første tråden er ferdig (en tråd av gangen). Til sist skal vi bruke Javas innebygde *synchronized* mekanisme som skal sikre at bare én tråd av gangen er inne i en av alle de metoder som er 'beskyttet av' synchronized-ordet i dette objektet. Hvis metodene er deklarert som static, vil denne synkroniseringen gjelde i alle objekter av denne klassen hvor denne beskyttelsen er brukt, men hvis metodene ikke er static, vil metode-beskyttelsen bare gjelde metodene i ett objekt ad gangen. Synchronized (som er et reservert ord i Java) kan beskytte en hel metode eller bare en blokk med setninger inne i en metode.

4.1 HVORDAN VIRKER SYNKRONISERING AV TRÅDER

Tråder er altså hver selvstendige sekvensielle programmer som kjører samtidig, vi sier i parallell, på én multikjerne PC.For å få adgang til CyclicBarrier, ReentrantLock som begge er klasser som man finner på Java.biblioteket ved å ha følgende import-setninger i toppen av program-koden :

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
```

Vi lager først ett objekt av en slik synkroniserings-klasse, og det er bare de trådene som kaller metodene i dette objektet som synkroniseres med hverandre (ved å kalle f.eks metoden await()).

I tillegg skal vi bruke synchronized som er bygget inn I programmeringsspråket Java og som kan sørge for at bare én tråd av gangen kan utføre en slik metode av alle de metodene i dette objektet med synchronized ordet foran seg i deklarasjonen. Vi kan bare synkronisere tråder i forhold til hverandre som bruker samme synkroniserings-objekt og alle objekter kan synkroniseres.
Et enkelt eksempel er at vi ønsker å debugge et parallelt program med tråder, og at vi ønsker å skrive ut verdiene av en variabel i en slik

De er altså alle tre mekanisme som greier å stoppe andre tråder midlertidig når en tråd skal skrive på data, en fil eller skjerm som er felles for alle trådene.

4.2 OM BRUK AV REENTRANTLOCK

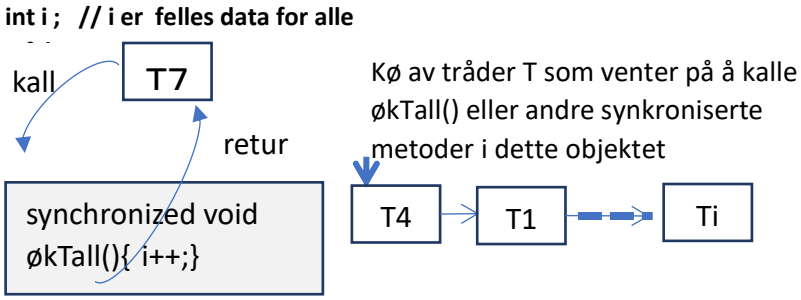
Den enkleste og raskeste låsen er ReentrantLock:

```
class X {
   private final ReentrantLock lock = new ReentrantLock();
   // ...

   public void m() {
     lock.lock();  // block until condition holds
     try {
       // ... method body
     } finally {
       lock.unlock()
     }
   }
 }
```

Program 3.1 *Kode for bruk av ReentrantLock som beskytter en metode m() fra å bli brukt av høyst en tråd.*

4.2 OM SYNKRONISERTE METODER

Synkroniserte metoder bruker låsen til det objektet de tilhører og sørger for at hvis mer enn én tråd kaller den, vil bare én slippe til av gangen og de andre må vente i en kø (fig 1.3). Kallende tråd får gjort seg helt ferdig og resultatet blir skrevet ned i hovedhukommelsen før neste tråd får utføre sitt kall.



Figur 3.1 *Synkroniserte metoder lar én tråd slippe til av gangen og køer opp andre tråder som evt. samtidig ønsker å gjøre et kall på synkroniserte metoder i alle objekter av denne klassen. Her holder tråd T7 på med et kall på økTall(), og mange andre tråder som har forsøkt å gjøre kall mens T7 holder på å få utført sitt kall, må vente i en kø. Når T7 er ferdig, slipper en av de andre løs fra køen og kan gjøre sitt kall, osv.*

Merk at hvis det er flere synkroniserte metoder i samme objektet, vil denne låsen i objektet sperre også for samtidige kall fra andre tråder på disse andre metodene. I ett objekt kan altså høyst en synkroniserte metode bli eksekvert av gangen. Merk at hvis man har flere objekter av den klassen hvor de synkroniserte metodene er, er det mulig å få utført en synkronisert metode samtidig av to eller flere tråder. Det kan være en utilsiktet feil, hvis kallene kommer til ulike objekter av denne klassen.
Vi kan først prøve å kjøre programmet i 3.1. med alle de 4 ulike definisjonene av metoden økTall(). Først spør den systemet om antall kjerner i maskinen og skriver det ut. Deretter lager den et objekt av

klassen Parallell og så leser den inn fra kommandolinja hvor mange tråder den skal starte om hvor mange ganger hver av dem skal øke heltallet tall med 1.Vi bruker her en long, 64 bit heltall for variabelen tall fordi summen av antall opptellinger (antGanger*antTråder) kan bli større enn største verdi for et 32 bit heltall. Vi kaller så metoden utfør() i dette objektet p av klassen Parallell.
Merk at:

Hvert objekt som skapes med new har en lås. Det er den låsen som nyttes ved kall på alle de synkroniserte metoder i det samme objekt som metoden er i. Kall på en synkronisert metode (den samme metoden eller en av de andre) fra en annen tråd i det samme objektet vil da bli låst fordi to slike metodekall nytter den samme låsen – dvs. det samme objektet. Når det første kallet er ferdig, slipper en av de andre trådene til med sine kall,… osv.

```
import java.util.*;
import java.util.concurrent.*;

/** Start >java Parallell <ant tråder> <ant ganger i løkke> */
class Parallell{
      long  tall=0;   // Sum av at 'antTråder' tråder teller opp denne
      CyclicBarrier b ;  //sikrer at alle er ferdige før vi tar tid og
sum
      long antTråder, antGanger ; // Etter summering: riktig svar er
                                  // antTråder*antGanger

     void utskrift(double tid) {
        System.out.println(«Tid «+antGanger+» kall * «+
          antTråder+» Traader =»+Format. align(tid,9,6)+ « sek,\n sum:»+
          tall +», tap:»+ (antTråder*antGanger -tall)+» = «+
          Format.align( (antTråder*antGanger - tall)*
          100.0/(antTråder*antGanger),5,1)+»%»);
     } // end utskrift

   synchronized void økTall(){ tall++;}         // 1)
   //  void økTall() { tall++;}                 // 2)

    public static void main (String [] args) {
       int antKjerner = Runtime.getRuntime().availableProcessors();
       System.out.println("Maskinen har "+ antKjerner + " kjerner.");
       Parallell p =  new Parallell();
       p.antTråder = Integer.parseInt(args[0]);
       p.antGanger = Integer.parseInt(args[1]);
       p.utfør();
     } // end main

     void  utfør () {
        b = new CyclicBarrier((int)antTråder+1);  //+1, også main venter
        long t = System.nanoTime();               // start klokke
        for (int i = 0; i< antTråder; i++)
              new Thread(new Para()).start();
        try{
             // main tråden venter
             b.await();
        } catch (Exception e) {return;}
              double tid = (System.nanoTime()-t)/1000000000.0;
              utskrift(tid);
       } // utfør


 class Para implements Runnable{
             public void run() {
                     for (int i = 0; i< antGanger; i++) {
                          økTall();
```

```
                    }
                    try {  // wait on all other threads + main
                       b.await();
                    } catch (Exception e) {return;}
            } // end run

            //    void økTall() { tall++;}              // 3)
            //    synchronized void økTall(){ tall++;}  // 4)
     } // end class Para
}// END class Parallell
```

Program 3.2. *Et program som viser både riktig og gal bruk av låser i Java. Vi ser 4 ulike plassering av en metode økTall() – kommentert med 1), 2) 3) og 4). Bare 1 er riktig. Inndata fra kommandolinja er hvor mange tråder vi vil starte og hvor mange ganger hver av disse trådene vil telle opp en felles variabel (long tall) i klassen Parallell. Hvis man fjerner kommentarmarkeringen // for én av kallene på økTall() vil programmet kunne kompileres og kjøre. Bare én av disse plasseringene er riktig. Alle de feilaktige plasseringene av metoden 'økTall()' vil som sluttresultat få alt for liten samlet sum i tall. Kjører vi et feilaktig program flere ganger med samme parametre vil det også nesten alltid gi ulike svar; typisk for synkroniseringsfeil.*

Vi kan lett gjøre den feilen at vi skaper flere objekter, og dermed flere låser. En tråd som kaller en synkronisert metode vil låse med den låsen som er i det objektet som utfører metoden. Program 1.2 viser et eksempel på en slik feil. Hvis vi 'av-kommenterer' plassering 4) og nytter den ser vi at vi får mange feil når vi kjører programmet (med 1000 eller flere oppdateringer). Grunnen til dette er at vi har laget en lås for hvert av tråd-objektene av klassen Para. Nå vil de synkroniserte variablene definert på denne måten, låses bare de kallene som nytter samme lås. Men siden hver tråd har sitt objekt og sin lås, vil ingen av trådene greie å låse ute de andre trådene – fordi de bruker hver sin lås.

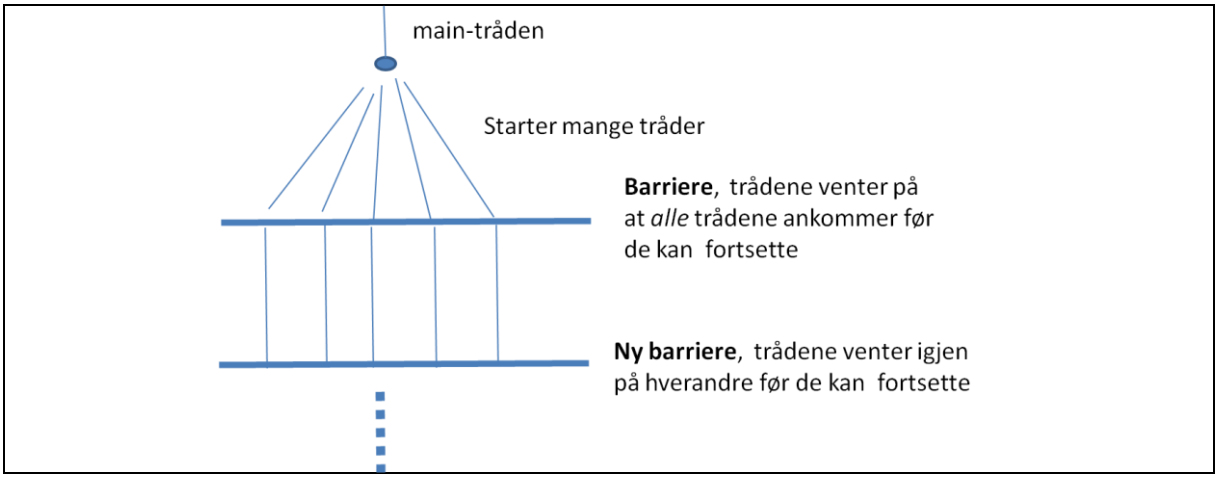
	1	2	3
Svar **uten** synchronized **3)**	7 112 531	5 911 630	6 169 492
Svar **med** synchronized **1)**	10 000 000	10 000 000	10 000 000

Tabell 3.1 *Tre kjøringer hvor vi starter 100 tråder som hver forsøker å øke variablen i 100 000 ganger med 1, dvs. i skulle da bli = 10 mill. Vi ser at uten at økTall() er riktig synkronisert, får vi mange feil og ulikt svar i hver kjøring.*

Hvis vi derimot nytter plassering 1) for den synkroniserte metoden økTall(), ser vi at den er inne i ett objekt av klassen Parallell. Det er bare dannet ett objekt av denne klassen, og alle kall på økTall()nytter da samme lås, og alt går da bra uansett hvor mange tråder og kall på økTall()vi har. Plasseringene 2) og 3) går også galt fordi det ikke brukes noen låser, tilsvarende som plassering 4) med mange opptellinger som går tapt. Alt fra 0,001 % til over 90 % av summen mangler. Resultatene varierer også fra kjøring til kjøring med samme parametere. Grunnen til at vi ikke alltid får tapte oppdateringer hvis vi har få kall på økTall() per tråd, er at hver tråd vi da starter, greier å gjøre seg ferdig før neste tråd starter. Vi får da ikke et skikkelig parallelt program, men at trådene utføres etter hverandre, sekvensielt.

5 BARRIERE SYNKRONISERING

Ikke alle beregninger kan greit eller effektivt løses med synkroniserte metoder. Mange beregninger kan parallelliseres ved at man deler dem opp i flere trinn. Hvert trinn gjøres i parallell av et antall tråder, men alle trådene må være ferdig med ett trinn før beregningene i neste trinn kan begynne. Da kan alle trådene fortsette med del to av beregningen, og da vet de at de kan lese hva de andre trådene skrev i forrige trinn av beregningen. Kanskje er det mange slike trinn i beregningene. Et viktig spesialtilfelle er at vi ikke trenger å dele opp selve beregningen i flere trinn, men at vi vil at hovedtråden, dvs. den tråden som programmet starter med i main, skal vente til alle trådene den har startet er ferdige med beregningene. Først da kan hovedtråden presentere resultatet til brukeren.



Figur 5.*1* *Programmet starter alltid først bare en tråd – maintråden. Den lager ett objekt b av klassen CyclicBarrier og et større antall tråder. Hvis barrieren er laget for **k** tråder, så vil alle tråder, også evt. maintråden, vente hvis de sier b.await() inntil **k** tråder har sagt b.await(). Da slipper alle de **k** trådene løs og kjører videre.*

For å få til en slik ventemekanisme for k stk. tråder, lager vi et objekt av klassen CyclicBarrier, og parameteren er antallet tråder som den skal køe opp; og når den siste melder seg skal alle trådene igjen slippes løs.

```
import java.util.concurrent.*;

CyclicBarrier b = new CyclicBarrier (antTråder);

< I trådene vil vi finne følgende kode når vi skal vente
i 'b' på at alle de andre trådene også er ferdige med
beregningene sine:>

     try {
            b.await();
     } catch (Exception e) {….}
```

Program 5.1. *Et program som skisserer riktig bruk av CyclicBarrier i Java.*

Vi vil i neste programeksempel se at vi har en parameter som er 1 større enn antall tråder vi lager, fordi vi bruker den sykliske barrieren b til at alle trådene og main-tråden venter på hverandre. Grunnen til at det heter CyclicBarrier er at når så mange tråder som den er spesifisert for har ventet og blitt sluppet fri, kan den uten videre motta det samme antallet tråder til ny runde med venting og

frislipping uten ny initialisering (den er straks gjenbrukbar). Hvis trådene har flere trinn hvor de må vente på hverandre, har vi gjerne to CyclicBarrier – én som settes opp med antTråder og som trådene bruker seg imellom, og én som initieres med antTråder + 1, som trådene venter på når helt ferdige og som main-tråden også har lagt seg til å vente på etter at den startet alle de andre trådene. I main-tråden vet vi da at når den slipper løs, har alle trådene blitt ferdige med koden sin.

Husk at her gjelder også det første punktet om synkronisering. Det å kalle på await() på en barriere sørger ikke bare for at alle venter, men også at alle etterpå kan se alt hva de andre skrev på felles variable før await()-kallet.

5.1 ET PROGRAM SOM BRUKER CYCLICBARRIER OG BEREGNER MAX-VERDIEN I EN ARRAY

```
import java.util.*;
import java.util.concurrent.*;

/** Start >java   FinnMax2 <ant tråder>  */
class FinnMax2{
      int[] a,lokalMax;  //finn max verdi i a[]
      CyclicBarrier b;   // sikrer at alle er ferdige før vi tar tid og
                        //  sum
      static int antTråder, ant;

      public static void main (String [] args) {
               antTråder = Integer.parseInt(args[0]);
               new FinnMax2().utfør();
       } // end main

      void  utfør () {
                  a= new int[antTråder*antTråder]; // større problem
                  ant= a.length/antTråder;   // antall elementer per tråd
                  lokalMax = new int [antTråder];
                  Random r = new Random(1337);
                  for (int i =0; i< a.length;i++) {
                        a[i] = Math.max(r.nextInt(a.length)-i,0);
                  }
                  b = new CyclicBarrier((int)antTråder+1); //+1, også main
                  int totalMax = -1;
                  long t = System.nanoTime();       // start klokke
                  for (int i = 0; i< antTråder; i++) {
                          new Thread(new Para(i)).start();
                  }

                  try{ // main venter på Barrieren b
                         b.await();
                  } catch (Exception e) {return;}

                 // finn den største max fra alle trådene
                    for (int i=0;i < antTråder;i++)
                      if(lokalMax[i] > totalMax) totalMax = lokalMax[i];

                System.out.println("Max verdi parallell i a:"+totalMax +
                  ", paa: "+((double)(System.nanoTime()-t)/1000000.0)+
                   " millisek.");

              // sammenlign med sekvensiell utføring av finnMax
              t = System.nanoTime();
              totalMax = 0;
              for (int i=0;i < a.length;i++)
```

```
            if(a[i] > totalMax) totalMax = a[i];
          System.out.println("Max sekvensiel:"+totalMax +", paa: "+
             ((double)(System.nanoTime()-t)/1000000.0)+ " millisek.");
      } // utfør

     class Para implements Runnable{
           int ind, minMax = -1;
           Para(int i) { ind =i;} // konstruktor

           public void run() { // Det som kjøres i parallell:
               for (int i = 0; i< ant; i++) {
                     if (a[ant*ind+i] > minMax) minMax = a[ant*ind+i];
               }
               lokalMax[ind] =minMax;  // levér svar

               try {  // wait on all other threads + main
                    b.await();
               } catch (Exception e) {return;}
             } // end run
       } // end class Para
}// END class Parallell
```

Program 5.1. *Et program som viser riktig bruk av CyclicBarrier i Java. Parameter til programmet er antall tråder, og det lages et array a[] som er antTråder*antTråder lang med tilfeldig positivt innhold. Vi starter så antTråder i parallell som finner maksimalverdien i hver sin del av a[], og tråd nr i legger sitt svar inn i lokalMax[i]. Hovedprogrammet som venter på den sykliske barrieren kan, når alle trådene er ferdige, selv gå gjennom lokalMax[] og finne totalMax-verdien. Vi skriver ut denne og tidsforbruket. Som sjekk går vi så sekvensielt gjennom a[] og skriver ut den max-verdien vi da finner og tidsforbruket for sekvensiell gjennomgang til sammenligning.*

Vi ser at dette er et program som greit parallelliserer beregningen av max-verdien i en array, men vi ser også av tidene som programmet skriver ut, at den parallelle beregningen tar ca. 50-100 ganger så lang tid som bare å lese gjennom arrayen sekvensielt fra start til slutt for beregningen av maksverdien. Dette eksemplet lærer oss forhåpentligvis bruk av en syklisk barriere, men også at parallellisering av svært enkle oppgaver hvor vi bare ser på hvert dataelement én eneste gang i beregningene, er det ingen vits i å parallellisere. Den ekstra tiden det tar å starte og stoppe tråder tar da langt lenger tid enn selve beregningene. Vi skal nå se på et problem, sortering, hvor parallellisering lønner seg, i alle fall hvis vi skal sortere mer enn 100 000 tall.

6 GENERELT OM PARALLELLISERING AV ALGORITMER, DEL 2

Her er en skisse av de stegene vi vanligvis foretar når vi lager en parallell algoritme for å løse ett problem.

1. Start med et vel testet sekvensielt program som løser problemet.
2. Del opp problemet i flere mindre deler som kan løses hver for seg.
   Vanligvis vil man søke å dele dataene i like store deler, ofte langt flere enn du har kjerner – f.eks 20* antKjerner. Grunnen til dette er at en tråd som løser et slikt delproblem kan få ventesituasjoner, og da er det greit at en annen tråd er i stand til å eksekvere. Imidlertid må det advares mot alt for mange tråder. Det tar tross alt noen få milliekunder å skape og starte en tråd. Vi kan godt bruke 10-500 tråder for å løse et problem, men ikke mange 10-tusner.
3. Start en tråd for hver av disse delene av problemet.
   Dette gjør vi hvis ikke oppdelingen og starting av tråder er avhengig av hvilke, og hvor mye data vi har. Da skjer en kombinasjon av oppsplitting av problemet og start av tråder samtidig under eksekvering – pkt. 2 og 3. kombinert.
4. La hver tråd løse en slik del.
   Vanligvis vil enten den samme, eller en lett modifisert versjon den sekvensielle algoritmen nyttes for hver slik del. Ofte erstattes da rekursjon med tråder. Felles data som flere tråder skriver på samtidig, beskyttes med synkroniserte metoder.
5. Vent til alle trådene er ferdige – f.eks med en syklisk barriere.
6. Kombiner del-svarene til en løsning på hele problemet. Av og til er det ikke nødvendig, men ofte er det slik at det er noen avsluttende beregninger.
7. Når du skal debugge et slikt parallelt program og f.eks. ønsker å skrive ut verdier av en eller flere variable i de ulike trådene, virker det dårlig å nytte System.out.println(String s) fordi den ikke er synkronisert og du vil oppdage at utskriften fra de ulike trådene blander seg på skjermen i en ikke lesbar mølje. Dette løser du enkelt ved å lage flg. variant av println:

   synchronized void println(String s){System.out.println(s);}

   Da vil Javas synkronisering sørge for at utskriften fra de ulike trådene *ikke* blander seg.

Pkt 2, 4 og delvis 6 er de vanskeligste og vil kunne variere fra problem til problem, pkt.3 og 5 går greit.

7 OM PARALLELLISERING MED STREAMS

I Java 8 og senere versjoner, nå er det innført begreper som stammer mer fra funksjonell programmering – her spesielt lambda-uttrykk og strømmer (streams). Grunnen til å nevne det her er at her synes det å være er programmerings-vennlig måte å parallellisere et program . Først en kort innføring i selve begrepene: lambda-utrykk og sekvensielle strømmer og deretter et eksempel på en vellykket parallellisering med parallelle strømmer, og til sist kommentarer til når denne mekanismen er egnet til parallelisering.
(Kan du vanlige sekvensielle strømmer og lambda-uttrykk kan du hoppe over avsnittene 7.1-7.5.)s

7.1 Lambda-uttrykk og strømmer (streams)

I Java 8 ble det innført et nytt nytt begrep som kalles *lambda-uttrykk*. Enkelt sagt er dette en kompakt måte å skrive (enkle) metoder som vi skal utføre. Kompilatoren *javac* finner da ut hvilke typer parametre vi har i *lambda-metoden* slik at vi kan slippe å skrive det, og mye annet vi kanskje ellers må skrive for å lage en metode. Koden vår blir kortere og mer lettlest.

Vi skal her gå gjennom hvordan *lambda-uttrykk* skrives, deklarasjoner og bruk. Slike uttrykke egner seg vel mest for små korte metoder som kan skrives på noen få linjer. Alt vi kan gjøre med *lambda-uttrykk* kan vi også få til med vanlige metode-deklarasjoner og bruk, men riktig brukt er *lambda-uttryk* mer elegant, lettere forståelig og gir brukere langt mindre å skrive særlig når *lambda-uttrykk* kombineres med *streams*.

7.2 Hvordan skrive lambda-uttrykk

Det er to måter å skrive lambda-uttrykk:

(parametre) -> uttrykk **eller** (parametre) -> { setninger;}

Begge tar parametre spesifisert på venstre side og bruker den på høyre side til å evaluere et uttrykk eller utføre setningene i klammeparentesen. Et lambda-uttrykk kan returnere en verdi eller bare gjøre noe.

Her er eksempler på lambda-uttrykk:

```
1. () -> 7              // har ingen parameter og returer 7
2. x -> 2 * x           // en parameter (x) og returnerer den doble verdien
3. (x, y) -> x – y      //tar to tall inn, returnerer differansen
4. (int x, int y) -> x + y          // tar to int inn, returnerer summen
5. (String s) -> System.out.print(s) // Tar en String s og printer den
```

Følgende er valgfritt når vi skriver *lambda-uttryk* :

- Type-deklarasjoner på parametre (kompilatoren finner det ut)
- Parentes rundt parametre hvis vi har bare en parameter.
- Krøll-parenteser {} rundt høyresida hvis det bare har en setning.
- Bruke av *return*-ordet i høyresida. Kompilatoren returnerer bare siste verdi beregnet, men utelater vi *return* må vi ha krøll-parenteser.

7.3 Hvordan lages lambda-uttrykk

Når vi lager et *lambda-uttrykk,* må vi ha et grensenitt med en metode som har samme antall og typer på parametrene som det vi trenger i vårt *lambda-uttrykk.* Hvis du enda ikke har lest kapittelet om grensesnitt, så er *grensesnitt* enkelt forklart en java – konstruksjon som ser ut som en meget forenklet klasse-deklarasjon. Et grensnitt begynner med ordet *interface* (i steden for class) og inneholder bare spesifikasjon av metoder (deres navn, parametre med typer og metodene returverdi, men *ikke* kode inne i metodene). Her er tre eksempler du selv kan skrive i ditt program:

```
interface  Calc{
      double beregn (int x);
}
interface  Hei{
      void si (String s);
}

interface  Matte{
      int oper (int x, int y);
}
```

Merk at grensesnitt for å lage *lambda-uttrykk,* kan bare har én metode. (Java-bibliotekene inneholder også mange slike grensesnitt som kan brukes til å lage *lambda-uttrykk,* men du kan like godt deklarere selv de grensesnitt du trenger):

I programmet ditt kan du så lage flere ulike *lambda-uttrykk* fra samme metode i samme grensesnitt når de bare har samme antall og type av parametre og samme returverdi.

7.4 Hvordan bruke Lambda

Før vi kan bruke et *lambda-uttrykk* må vi altså koble det uttrykket vi vil bruke med et grensesnitt med samme antall og type parametre. Vi navngir da et *lambda-uttrykk* (egentlig navngir vi et objekt av en klasse som lager (implementerer) som den metoden som er i grensesnittet) slik :

```
Hei joa  = (s) -> System.out.println("Jeg sier:"+ s);
```

Og vi kan kjøre denne metoden slik (i en av våre vanlige metoder som main):

```
joa.si(« Lambda er bra»);  // ut: Jeg sier: Lambda er bra
```

Her er er flere eksempler:

```
Calc areal = x -> x*x*3.14159/4;
Calc radius = x -> x*1.0/2;
Matte mult = (x,y) -> { return x*y;};
```

Og det kan brukes slik:

```
Scanner keyboard = new Scanner (System.in);
System.out.print ("Gi et heltall: ");
num = keyboard.nextInt();

System.out.println(" Sirkel med diam:"+num+" m har areal:"
 + areal.beregn(num)+" m2, og radius:"
 + radius.beregn(num)+"m");
System.out.println("3*7=" + mult.oper(3,7) );
joa.si(" - dette er bra");
```

Vi ser at grensenittet Calc har to ulike implementasjoner (areal og radius). Kjører vi programmet over får vi:

```
Gi et heltall: 137
Sirkel med diam:137 m har areal: 14741.1256775 m2, radius:68.5m
3*7=21
Jeg sier: - dette er bra
```

For å oppsummere: *lambda* gir ikke noe nytt, men du skriver mindre kode for å få det utført. I neste delkapittel ser vi hvordan *lambda-uttrykk* med brukes sammen med strømmer.

7.5 Strømmer (streams) fra mengder

Strømmer i Java er kort fortalt å kunne stille spørsmål som minner om SQL spørsmål fra database-verden. Vi har en startmengde i Java (List, ArrayList, array, HashMap,..) som vi først omgjør til en strøm av enkeltobjekter og fra denne lager vi en ny mengde (svar-mengden), som er de fra startmengden som tilfredsstiller de kravene vi kommer med i den strømmen vi lager. Ta et eksempel fra programmet nedenfor om personer, deres navn, kjønn, inntekt osv. Vi kunne være interessert i å vite hvilke

personer som tjener over 1 mill kr. Da søker vi ut en ny mengde. Vil vi bare ha navnet til én person med så stor inntekt, søker vi etter en string. Vi kan også være interessert i bare å vite hva er snittinntekten til de som tjener over 1 mill. kr. Vi søker da et tall. Vi kan også være interessert i finne kvinners inntekt. Vi kan altså både søke ut en ny mengde, men også tall, Stringer, ol.

```
import java.util.*;

class Person{
    String navn; String kjonn ; int alder; int lonn;
    Person (String na, String kj, int ald, int ln) {
           navn = na; kjonn = kj; alder =ald; lonn = ln;
    }
 }

 public class StreamEks{
     ArrayList <Person> alle = new ArrayList <Person> ();

     public static void main (String [] args) {
           new StreamEks().doIt();
    }
    void doIt() {
       alle.add(new Person("Ola",  "M", 55,1200000));
       alle.add(new Person("Kari" ,"F", 44, 600000));
       alle.add(new Person("Jonas","M", 65, 110000));
       alle.add(new Person("Tora", "F", 12,  10000));
       alle.add(new Person("Arne", "M", 69, 2000000));

      // 1) skriv alle som tjener mer enn 1. mill
        alle.stream()
         .filter(s-> s.lonn > 1000000)
         .forEach(p->System.out.println(p.navn+
                      " tjener kr. "+ p.lonn+" per år"));

      // 2) Finn person med størst inntekt
        int ml =
          alle.stream()
           .mapToInt(s -> s.lonn)
           .max()
           .getAsInt();

     // 3) finn gjennomsnittsinntekt for alle
          alle.stream()
           .mapToInt(p -> p.lonn)
           .average()
           .ifPresent(t -> System.out.println("snitt lønn= "+ t));

      // 4) Beregn snitt inntekt for de med lønn > 1.mill
           alle.stream()
            .mapToInt(p -> p.lonn)
            .filter (r -> r > 1000000)
            .average()
            .ifPresent(t -> System.out.println(
              "Snitt de over 1.mill = " +t+ " kr."));

     // 5) Beregn snitt kvinners lønn
          double kvinneLønn =
             alle.stream()
            .filter(p -> p.kjonn == "F")
            .mapToInt(p -> p.lonn)
            .average()
            .getAsDouble();

           System.out.println("Gjsnitt kvinnelønn er kr."+
              kvinneLønn+", max lønn er:"+ ml);
```

```
        // 6) Finn første kvinne med lønn < kr. 100 000
               alle.stream()
                  .filter(p -> p.kjonn == "F")
                  .filter (r -> r.lonn < 100000)
                  .findFirst()
                  .ifPresent(t -> System.out.println(
                            "Lavlønnet kvinne er:"+t.navn));

      } // end doIt
} // end StreamEks
```

Resultatet fra kjøring er:

```
Ola tjener kr. 1200000 per år
Arne tjener kr. 2000000 per år
snitt lønn= 784000.0
Snitt de over 1.mill = 1600000.0 kr.
Gjsnitt kvinnelønn er kr.305000.0, max lønn er:2000000
Lavlønnet kvinne er:Tora
```

Forklaring til alle eksemplene er at vi omgjør mengden vår først til en strøm (stream()) av enkelt-elementer, her Person-objekter. Videre i strømmen er det enten funksjoner som begrenser hvilke objekter som kommer videre (eks. filter()) , funksjoner som omgjør et objekt til en annen bestemt type (eks MapToInt()) og funksjoner som ber om alle (som max(), average(), sum(), forEach()) eller som bare ber om ett eksemplar (findFirst()). En slik strøm drives fremover ved det stadig bes om nye elementer, og hvis den funksjonen som er sist eller nest sist i har blitt tilfredstillet, som i eks 6 at vi finner første kvinne med lav lønn (under 100 000). Se dokumentasjonen til grensesnittene java.util.stream og java.util.collection i Java-dokumentasjonen.

Grunnen til at at vi ikke behøver å skrive egne grensesnitt her er at de bibliotekene vi her bruker i java.util har deklarert det vi trenger med riktige typer på parametre og returverdier. Vi ser også at vi må prøve oss frem når vi tak et gjennomsnitt og må i Eks.3 konvertere det som 'flyter nedover strømmen' før vi kan ta average(), eller i Eks. 5 må vi også konvertere resultatet til en ekte double-verdi før vi kan gjøre tilordning til en enkel double-variabel.

7.5 Parallelle strømmer

Her er en meget enkel metode som avgjør om parameteren er primtall eller ikke. Dette er ikke en effektiv metode (Eratosthenes sil er langt raskere) og er bar tatt med her som eksempel på en metode som bruker en del tid – den dividerer parameteren med 2 pluss alle oddetall < enn kvadratroten av parameterverdien. (Dette eksempelet har jeg fått av prof. Peter Sestoft, IT-Universitet i København og omarbeidet noe.

```
      /* returnerer true hvis p er primtall */
      private static boolean isPrime(int p) {
            if (p % 2 == 0) return p == 2;
            int k=3, k2=9;

            while ((p%k != 0) && k2 <= p ){
                  k+=2;
                  k2=k*k;
            }
            return  k2 > p;
      }// end isPrime
```

Vi kan nå lage en sekvensiell strøm og parallell strøm som bruker isPrime og senere et sekvensielt og parallelt program slik vi hittil har lært i IN3030 for å sammenligne effektiviteten av slike strømmer.

```
//    Sekvensiell strøm:
          int antall=
             IntStream.range(2,n)
             .filter (i-> isPrime(i))
             .count();

//        Parallell strøm:
          int  antall=
               IntStream.range(2,n)
               .parallel()
               .filter (i-> isPrime(i))
               .count();
```

Legg merke til at at det eneste som settes inn i programmet for den sekvensielle strømmen er en kall på funksjonen som parallellisere strømmen av tall som kommer fra IntStream.range(2,n) som genererer tallene fra og med 2 til (men ikke med) n. Siden dette kjøres på 4 kjernes CPU som på grunn av at hver av disse er hyperthreaded, så vil operativsystemet si at den har 8 kjerner. Legg merke til hvor enkelt det er å parallellisere en strøm, og siden utviklerkostnadene (lønn) er langt viktigere enn kostnadene til en server, vil dette være en kost-effektiv måte å paralleliser problemer som er enkelt parallelliserbare.

For å lage et sekvensiell vanlig program som gjør det samme som strømmeløsningene må vi skrive tilsvarende funksjoner. IntStream.range(2,n) programmeneres som en vanlig for-løkke. Så

må vi lage en metode som teller opp hvor mange primtall vi finner < n (tilsvarer funksjonen filter () og count i strømmeløsningene:

```
/* returnerer anttall primtall p < n */
int tellPrim (int n) {
    int count = 0;
    if ( n>2 ) count = 1;// because of 2

    for (int i =3; i < n; i+=2)
        if ( isPrime(i)) count++;
    return count;
} // end tellPrim
```

For å lage et parallelt program som løser dette problemet må vi i tillegg lage en metode som teller hvor vi finner antall primtall mellom a ≤ p <b. Vi deler da problemets data mellom de ulike trådene:

```
/* parallell løsning, returnerer antall primtall p for a<= p <b */
 int tellPrim (int a, int b) {
     int count = 0, size = b-a;
     if (size <2) {
           if (a == 1)          return 0;
           else if (isPrime(a)) return 1;
     } else {
          if (a%2 == 0) a++;   // test only odd numbers

          for (int i =a; i < b; i+=2)
               if( isPrime(i)) count++;
     }

     return count;
} // end tellPrim parallel
```

Så kommer koden til Arbeiderne som finner antall primtall i hver sin del av talllinjen. I tillegg må ha kode som starter trådene og som så venter på at alle Arbeider-objektene terminer og som så summererer verdiene i 'tell'-arrayen.

```
class Arbeider extends Thread { //implements Runnable {
            int ind;
            int left,right;
            // lokale data og metoder

            Arbeider (int in, int left, int right) {
                     ind = in;
                     this.left = left;
                     this.right= right;
             }

             public void run( ) {
                     tell[ind] = tellPrim(left,right);
             } // end run

 } // end indre class Arbeider
```

Legg merke til at vi skriver vesentlig mer kode vi får når vi lager parallelle programmer i IN3030 enn man gjør med parallelle strømmer. Den eneste funksjonen vi skriver med parallelle strømmer er: 'isPrime(int p). Siden lønningene til systemutviklere er den største utgiften i ethvert prosjekt favorisere det klart strømmer.

Tidsforbruket for de 4 algoritmene vi testet (sekvensielt 'vanlig' program, sekvensiell strøm, vanlig' parallelt program og parallelt strøm) for ulike verdier av n =1,,1 000 000 er (merk logaritmiske verdier på begge akser fordi verdiene varierer så mye):

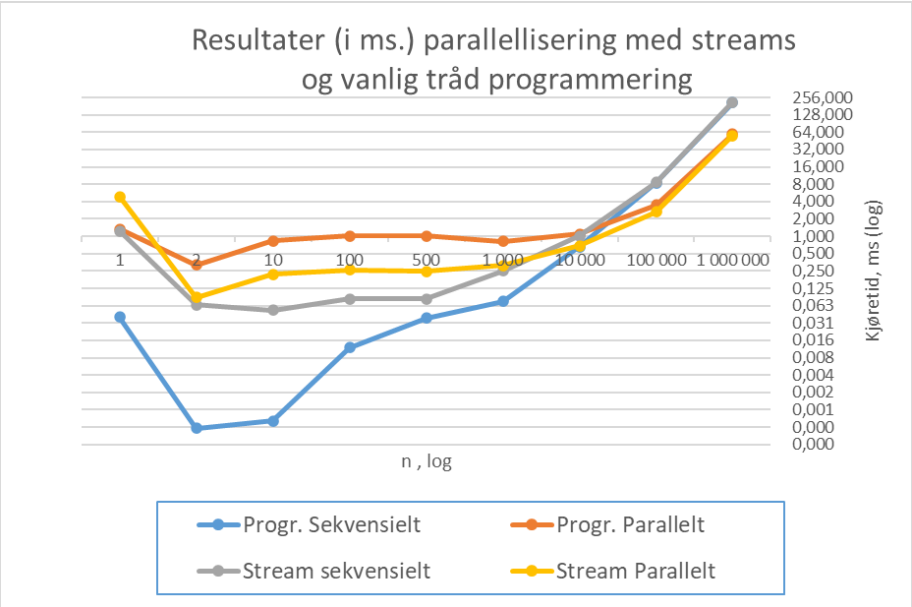


Fig 7.1 *Eksekveringstiden for de 4 ulike metodene ('vanlig' sekvensiell og parallell løsning) og strømme løsning(sekvensiell og parallell).*

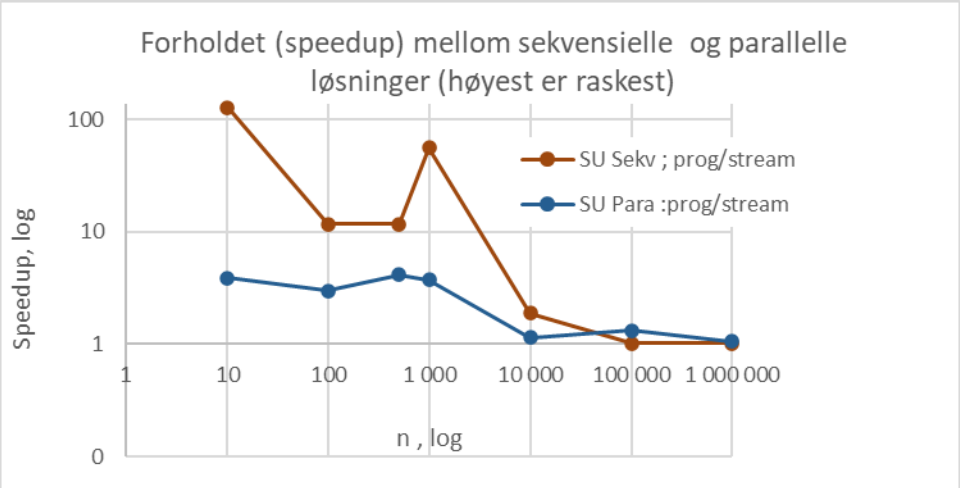


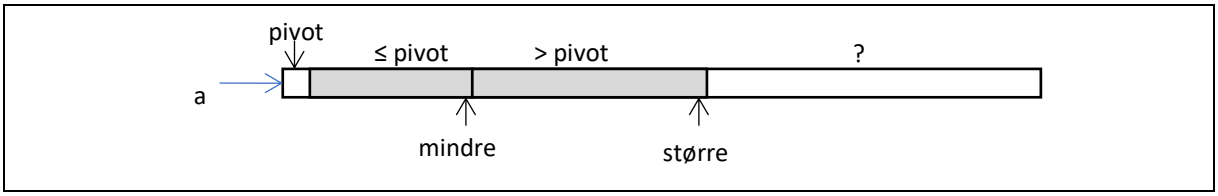
Fig 7.12 *Forholdet mellom 'vanlig' program og strømme-løsning (speedup sekvensiell og parallell).*

Vi ser av fig. 7.1 og 7.2 at det sekvensielle vanlige programmet er over 100 x raskere enn en sekvensiell strøm for små verdier av n, men når kjøretidene er >1 ms for n > ca 20 000, forsvinner denne klart raskere faktoren, og at de parallelle versjonene av både strømmer og vanlig program er om lag rike raske og begge de to parallelle løsningene er nesten 4 ganger raskere enn de sekvensielle løsningen – fordi vi har 4 kjerner. Konklusjonen må bli at parallelle strømmer må foretrekkes der de kan brukes på store strømmer, og selv om det er 'mye' langsommere for mindre vedier av n, antall objekter i strømmen er kjøretidene så små at det vel ikke er viktig. Fordelen med strømmer er at mye triviell kode er fjernet og er programmeringstiden er vesentlig kortere og enklere.
En klar begrensning med strømmer er at man kan ikke endre mengden strømmen starter med under kjøring og at man parallelliserer hvert element i strømmen, noe som vel ikke er mulig med Eratosthnes algortme. Vi ser også at parallelliseringen i strømmer er per element i strømmen, mens vi parallelliser i IN3030 måten å programmere på gir hver tråd hver sin separate del av talllinja.

8 KVIKKSORT, PARALLELLISERING AV EN REKURSIV ALGORITME

Som nevnt ovenfor, er en av de gylne reglene i parallellprogrammering at før man skriver et parallelt program, lager man et godt testet sekvensielt program som løser problemet. Parallelliseringen er endringer til det sekvensielle programmet.

Vi skal da presentere først en sekvensiell sorteringsmetode KvikkSort (eng. QuickSort), laget av Tony Hoare i 1962. Den fikk en enkel utforming av Nico Lamuto som vi nytter her. Senere skal vi parallellisere denne- Idéen er enkel: Velg ut et element i arrayen, kalt pivot-elementet. Bytt om elementene i arrayen slik at alle elementer som er ≤ pivot kommer til venstre for alle de elementene som er > pivot. For å forenkle koden plasserer vi selve pivot elementet helt til venstre i den delen vi nå sorterer. Når vi er helt ferdige med sorteringen, plasseres 'pivot' elementet mellom de to delene, Arrayen a er da ikke ferdig sortert, men hvis vi nå for hver av de to delene gjør samme type oppsplitting *med nye valg av pivoter*, så kan de igjen oppsplittes gjentatte ganger til alle de mange delene til slutt har en lengde på 1 eller 0. Da er a[] sortert fordi ethvert element da står til høyre for et annet element som er mindre eller lik dette.



Figur 8.1 *Idéen bak Kvikksort. Vi velger først et vilkårlig element pivot, så bytter vi om på elementene i a[] slik at vi får de som er mindre eller lik pivot til venstre, så alle de som er større enn pivot til høyre og pivot midlertidig helt til venstre. Til slutt plasseres pivot mellom de to delene Dette gjentas på hver av de to delene, på hver av disses to deler igjen osv. til a[] er sortert.*

Her er skjelettet til programmet som gjør denne sorteringen med oppsplitting gjentatte ganger, og som måler tiden og skriver ut denne:

```
import java.util.*;

/** Sekvensiell implementasjon av Quicksort */
class SeqQuick {
      int [] a;

      /** bytter om a[i] og a[j] */
      void bytt(int [] a, int i, int j) {..}

      /** Innpakning for for sQuick  - enklere kall  */
      void sQuick(int [] a) { sQuick(a, 0,a.length-1);}

      /** Rekursiv, sekvensiell QuickSort av a[lav..høy] */
      void sQuick (int [] a,  int lav, int høy) {..  }

      /** Konstruktor, fyller a[0..n-1] med tilfeldige tall */
      SeqQuick(int n) {.. }

       /** Tar tider, kaller sQuick og gjør en enkel test */
       void utførOgTest() {.. }

       public static void main (String [] args) {
             new SeqQuick(Integer.parseInt(args[0])).utførOgTest();
       }
} //end SeqQuick
```

Program 8.1 *Skjelettet for programmet som starter i main, som først lager et objekt av klassen SeqQuick med kall på konstruktøren og deretter kaller utførOgTest.*

Koden for konstruktøren og utførOgTest:

```
SeqQuick(int n) {
      a = new int [n];
      Random r = new Random(1337157);
      for (int i =0; i< a.length;i++)
            a[i] = r.nextInt(a.length); // random fill >=0
 } // end konstruktør

 void utførOgTest( ) {
      long  t = System.nanoTime();       // start klokke
      sQuick(a);
      t = System.nanoTime()-t;
      System.out.println("Sekvensiell QSort av "
                         +a.length+" tall paa:"+
      ((double)(t)/1000000.0)+ " millisek.");

       // test
       for (int i = 1; i<a.length; i++) {
              if (a[i-1] > a[i] ) {
              System.out.println("FEIL  a["+(i-1)+"]:"
                              +a[i-1]+"a["+i+"]:"+a[i]);
              return;
       } }
 } // end utførOgTest
```

Program 8.2 *Koden for konstruktøren, som oppretter a[] og fyller den med tilfeldige tall <n; og* utførOgTest(), som tar tida med System.nanoTime() som gir tida i nanosekunder (milliardedels sekunder). *Den kaller så sQuick(), skriver ut tida i millisekunder og gjør en enkel test på om sorteringa gikk bra.*

Koden er forklart under kodelistingen. Testen som utføres er strengt tatt ikke god nok. En enkel og komplett test ville være å sortere de samme tallene med javas innbygde sorteringsalgoritme: java.util. Arrays.sort, og så sammenligne de to sorteringene element for element. Man kunne også ta tida på Arrays.sort og sammenligne tidene (se oppgave 1).

```
void bytt(int [] a, int i, int j) {
      int t = a[i];
      a[i] = a[j];
      a[j]=t;
  }// end bytt

  /** Rekursiv, sekvensiell QuickSort av a[lav..høy] */
   void sQuick (int [] a,  int lav, int høy) {
     int ind =(lav+høy)/2,
         pivot = a[ind];
     int    større = lav+1,  // hvor lagre neste '> piv'
            mindre =lav+1;  // hvor lagre neste '<=  piv'

     bytt (a,ind,lav);     // flytt 'piv' til a[lav] , sortér resten

     while (større <= høy) {
           if (a[større] < pivot) {
                 // a[større] er 'mindre' - bytt
                 bytt(a,større,mindre);
                 ++mindre;
           }
           ++større;
     } // end gå gjennom a[lav+1..høy]

     bytt(a,lav,mindre-1);     // Plassert 'piv'- mellom store og små

     if ( mindre-lav > 2)  sQuick (a, lav,mindre-2); // sorter <= pivot
     if ( høy-mindre > 0)  sQuick (a , mindre, høy); // sorter > pivot
  }// end sQuick
```

PROGRAM 8.2 KODEN FOR SELVE SORTERINGEN: METODENE SQUICK OG BYTT.

Bytt er rett fram kode. Koden til sQuick går i to faser. Først er det et valg av **pivot**-element og oppdeling av den delen av arrayen som pekes ut i kallet med lav og høy. Deretter kommer to rekursive kall – ett på de som er ≤ **pivot** og ett på den delen hvor de som er > **pivot** ligger. Selve logikken til oppdelingen illustreres av fig 1.6. Vi har to pekere i en løkke: **større**, som peker på den plassen vi vil ha *neste* element som er > **pivot**; og mindre som peker på den plassen hvor vi vil ha *neste* element som er **≤ pivot.** Variabelen **større** økes alltid med 1 i hver løkkegjennomgang hvor vi ser på elementet **a[større].** Er **a[større] ≤ pivot**, så bytter vi det med **a[mindre]** som jo peker på det elementet som er 'lengst-til-venstre' av de som er > **pivot.S**å øker vi **mindre** med 1.

Merk at vi plasserer **pivot** mellom de to delene når vi er ferdige. P**ivot** står da på sin endelige plass i sorteringen. Den skal aldri mer flyttes og er ikke med på den videre todeling (som egentlig da er en tredeling) av hver del som vi skal dele videre opp. Behandlingen av **pivot** er litt spesiell – først tar vi og setter den helt til venstre i **a[lav].** Så deler vi resten av arrayen i to deler, og så bytter vi **pivot**, som står på i **a[lav]** med det elementet som står lengst til høyre av de som er ≤ **pivot**: **a[mindre-1].** Grunnene til dette er to. Hvis **pivot** viste seg å være det største av alle elementene vi nå skal sortere, ville vi få en uendelig rekursjon hvis vi ikke gjør dette fordi vi ikke fikk delt opp i to deler, men i en. Den andre grunnen er at vi da plasserer **pivot** på sin endelige plass, og at summen av de delen vi sorterer videre er ett element mindre. Dette gjør at programmet vil terminere uansett hvor uheldige vi er i valg av **pivot**.

Etter at vi har byttet inn **pivot** mellom de to delene, gjør metoden kall på seg selv for de to delene til venstre og høyre for **pivot** hvis disse har større lengde enn ett element. Det er særlig denne kodedelen som blir annerledes i en parallell versjon av kvikksort vi skal se på i neste avsnitt.

Denne koden for Kvikksort er spesielt rask for alle små verdier av n, og like rask for større verdier av n som den innebygde sortingsmetoden **Arrays.sort(..)** i biblioteket **java.util**, som er en annen og mer komplisert koding av Kvikksort.

10 EN BLANDET PARALLELL OG REKURSIV KVIKKSORT

Grunnidéen i en parallell versjon av kvikksort er at vi bytter ut de rekursive kallene med at vi i stedenfor starter en ny tråd for hvert av de to kallene. Men som vi så av de to foregående eksemplene tar det en viss tid å starte og stoppe tråder, og sortering går meget fort (fig. 2.1). Vi må derfor lage en blandet algoritme som nytter tråder når vi f. eks deler opp en del av arrayen som er lenger enn 50 000 elementer, men som nytter rekursjon for kortere deler.
Skjelett-koden til programmet for parallell sortering en ganske likt det for sekvensiell kvikksortering:

```
import java.util.*;
import java.util.concurrent.*;

class ParaQuick {
      int [] a;
      int antTråder = Runtime.getRuntime().availableProcessors();
      final static int PARA_LIMIT = 50000;

      synchronized void  tellOppAntTråder () { antTråder ++;}

      void bytt(int [] a, int i, int j) {…}

      void pQuick(int [] a) {  pQuick(null,a, 0,a.length-1); }

      void pQuick (CyclicBarrier b,int [] a, int lav, int høy) {…}

      ParaQuick(int n) { … } // konstruktor

      void utførOgTest() {… }

      public static void main (String [] args) {
             new ParaQuick(Integer.parseInt(args[0])).utførOgTest();
      }

      class Para implements Runnable{
            int [] a; int lav,høy;CyclicBarrier b;
            Para(CyclicBarrier b, int []a,int lav,int høy) {
                  this.a=a;this.b=b;this.lav=lav;this.høy=høy;
                  tellOppNumThr();
               }
               public void run() {
                     pQuick(b,a,lav,høy);
               } // end run    }
} //end ParaQuick
```

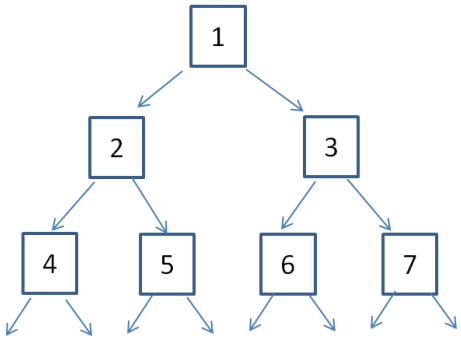
Program 10.1 *Skjelettkoden for parallell kvikksortering. Konstruktøren paraQuick har samme kode som konstruktøren SeqQuick i prog. 1.6. Også metoden bytt er identisk med den sekvensielle bytt.*

Vi ser at internt i klassen **ParaQuick** har vi i tillegg til arrayen **a**, en variabel **antTråder** som spør operativsystemet hvor mange kjerner vi har og da hvor mange tråder ***k*** vi kan starte i parallell på denne maskinen. Den vesentligste endringen i forhold til den sekvensielle versjonen, er at vi har innført en indre klasse **Para** som inneholder **run()** - metoden som er den koden vi skal utføre i denne tråden parallelt med de andre trådene. Den parallelle koden er et kall på **pQuick** for å få sortert den delen av **a[]** som denne tråden skal sortere.

```
void pQuick (CyclicBarrier b,int [] a, int lav, int høy) {
      int ind =(lav+høy)/2,
      piv = a[ind];
      int    større=lav+1,  // hvor lagre neste 'større enn piv'
      mindre=lav+1;         // hvor lagre neste 'mindre enn piv'
      bytt (a,ind,lav);        // flytt 'piv' til a[lav] , sortér resten

      while (større <= høy) {
         if (a[større] < piv) {
            bytt(a,større,mindre);
            ++mindre;
         }
         ++større;
      }

      bytt(a,lav,mindre-1);        // Plassert 'piv' mellom store og små

      if ( høy-lav > PARA_LIMIT){
         CyclicBarrier b2 = new CyclicBarrier(3);
         new Thread(new Para(b2,a,lav,mindre-2)).start(); //  <= piv
         new Thread(new Para(b2,a,mindre,høy  )).start(); //   > piv
         try {  // wait on own two calls to complete
                  b2.await();
          } catch (Exception e) {return;}
      } else {
          // korte arraysegmenter raskere med rekursjon
          if ( mindre-lav > 2) pQuick (null,a, lav,mindre-2);//  <= piv
          if ( høy-mindre > 0) pQuick (null,a, mindre, høy); //  > piv
      }

      if (b!= null) {
           try {  // signaliser til kallende tråd at denne er ferdig
              b.await();
           } catch (Exception e) {return;}
      }
  }// end pQuick
```

Program 10.2 *Sorteringsalgoritmen pQuick. Delingen av a[] er akkurat den samme som ved sekvensiell sortering. Det interessante er hvordan vi parallelliserer de to delene etter oppsplittingen. Hvis hele den delen vi skal sortere er større enn PARA_LIMIT, oppretter vi en syklisk barriere som skal vente på 3 tråder – de to nye trådene som startes og den tråden som skaper disse. Ventesetningen like etter at de to trådene som skapes er at den tråden som skapte disse to trådene, venter på at de begge er ferdige. Helt på bunnen av metoden ser vi en ny await() kall. Det er signalet til den tråden som kalte denne metoden, at denne tråden er ferdig – altså et signal oppover.*

Koden forklares i teksten under program 9.1. I oppgave 2 skal dere prøve ut og forklare tidsforbruket uten å kode dette med rekursiv løsning for kortere deler, men bare med tråder.



***Figur 10.2** Kall-treet. Parallell Kvikksort bruker enten tråder eller rekursjon for å løse problemet. Dette kan illustreres som et tre (med roten i toppen) av de ulike instansene av metoden pQuick hvor ett høyere nivå gjør to kall på neste nivå. Merk forskjellen på rekursjon og tråder. Starter vi tråder, vil de (grovt sett) starte i den rekkefølgen som de her er nummerert: først 1 som starter 2 og 3, så vil 2 starte 4 og 5, mens 3 vil starte 6 og 7, osv. Dette kalles bredde-først. Hvis det imidlertid dreier seg om rekursive kall, da vil 1 starte 2 som vil starte 4,…og først etter at alle kall som genereres av 4 og dets 'barn' har returnert til 2, vil 5 bli kalt, så 3,6 og 7. Dette kalles dybde-først traversering av treet.*

10.1 EFFEKTIVITETEN PÅ SEKVENSIELL OG PARALLELL KVIKKSORT

Vi testet de to versjonene av Kvikksort på to ulike maskiner, én med 8 kjerner og én med 64, og resultatene er referert i tabell 1.2 og 1.3.

	10	100	1 000	10 000	100 000	1 mill	10 mill	100 mill
Sekvensiell	0,01	0,05	0,70	13	18	89	963	11196
Parallel	0,01	0,06	0,84	16	24	67	356	3579
SU (Speedup)	1,00	0,83	0,84	0,81	0,75	1,32	2,71	3,12

***Tabell 10.2** Antall millisekunder det tar å sortere en array med tilfeldig innhold med den sekvensielle og parallelle versjonen av Kvikksort (parallelt for deler som er lengre enn 50 000) og Speedup (sekvensiell/parallell). Kjørt på en Intel i7 870, 3Ghz klokke med 8 kjerner.*

	10	100	1 000	10 000	100 000	1 mill	10 mill	100 mill
Sekvensiell	0,01	0,05	0,81	18	21	197	1413	16497
Parallel	0,01	0,06	0,99	21	56	106	1332	5025
SU (Speedup)	1,00	0,83	0,82	0,85	0,38	1,85	1,06	3,28

***Tabell 10.3** Antall millisekunder det tar å sortere en array med tilfeldig innhold med den sekvensielle og parallelle versjonen av Kvikksort(parallelt for deler som er lengre enn 50 000) og Speedup (sekvensiell/parallell). Kjørt på en maskin med 4 Intel Xeon prosessor L7555, 1.87Ghz klokke, totalt med 64 kjerner.*

Kommentarer til tabellene. Først tidene for sekvensiell utførelse. Vi ser at tidene er langt mer enn 20 ganger så lange når vi går fra å sortere 1000 til 10 000 tall (de burde bare vært litt mer enn 10-ganger) og tilsvarende videre til 100 000. Det kan forklares med at mesteparten av arrayen vi sorterer først ikke får plass i nivå 1 cachen, men i nivå2, og for 100 000 tall i nivå3 cachen og hovedhukommelsen som jo er mye langsommere.

Det beste vi synes å oppnå er at den parallelle versjonen går om lag 3 ganger så raskt som det sekvensielle programmet og ikke så mange ganger fortere som vi har kjerner. De parallelle

resultatene kan forklares ved flere faktorer, men det interessante spørsmålet er: Hvis vi har k kjerner, hvorfor går det ikke k ganger fortere?

Det er i hovedsak to grunner. Først er ikke parallelliseringen optimal. Den første oppdelingen av **a[]** i to deler skjer sekvensielt og like langsomt som den sekvensielle algoritmen. Først da starter vi 'bare' to tråder. Når hver av disse igjen deler opp hver sine deler i to, får vi fire tråder som er aktive (og ikke venter), osv. Vi får altså ikke brukt alle kjernene helt fra starten av programmet.

Den manglende parallelliseringen er imidlertid ikke alltid hovedforklaringen. I figur 19.2 ser vi en strek som forbinder hver prosessor med hovedhukommelsen. Det kalles en hukommelses- eller data-kanal, og kan ikke gå fortere enn hukommelsen kan operere. Vi husker også at hovedhukommelsen var mye langsommere enn hver kjerne. Det betyr kort sagt at det er en kø av kjerner på hukommelseskanalen for å lese og skrive i hovedhukommelsen, og det blir mye venting. Vi greier ikke å få trådene til å jobbe med full hastighet.

Lite køing på hukommelseskanalen har bare problemer som har lite data og som gjør mer arbeid med hvert dataelement enn de problemene vi har sett på her. Et slikt problem ("Selgerens rundreise") ser vi på i oppgave 3. Da vil data som det jobbes med i kjernen i stor grad være in cache-hukommelsene og antall lesinger og skriveoperasjoner i hovedhukommelsen vil ikke skje så ofte at det skaper kø på hukommelseskanalen. Parallelliseringen av kvikksorteringen kan imidlertid ikke sees på som mislykket – at store sorteringer går tre ganger så fort er klart noe man vil ønske i praksis.

11 AMDAHLS LOV

Vi ser av Kvikksort-eksempelet først hadde en sekvensiell del (dele arrayen i to) , og så kunne parallelliseringen begynne. Dette er generelt for mange algoritmer, at de har ofte først har en sekvensiell del. Et eksempel kan være at den sekvensielle delen tar 10% av tiden og de delene som kan parallelliseres tar da 90% av tiden når vi kjører alt sekvensielt. Vi ser da at det beste vi da kan oppnå av en parallell algoritme, er at den maksimalt vil gå 10 ganger så fort som den sekvensielle. Uansett hvor mange hundre- eller tusenvis av kjerner vi bruker på den parallelle delen, kan vi ikke få den til å gå raskere enn på 0,0 sekunder. Vi står da igjen med 10% av beregningene i den sekvensielle delen – altså maksimalt 10 ganger raskere enn en sekvensiell beregning..
Generelt, anta at vi har en algoritme som må utføre p % av algoritmen sekvensielt og at vi greier å få den delen av koden som kan parallelliseres til å gå k ganger raskere. Da er den maksimale hastighetsforbedringen S vi kan få:

$$S = 100/(P + (100-P)/K$$

Setter vi inn at den sekvensielle delen p = 10% og antall kjerner k = 1000, ser vi at S blir 9,9 – vi greier altså å gå 9,9 ganger fortere. Greier vi å få p ned i 1% blir S=91 med de samme forutsetningene. Lærdommen fra Amdahls lov er at før eller siden er det den delen som ikke kan parallellisere som vil dominere kjøretiden. Det er nesten alltid sånn at noe må gå sekvensielt – f.eks skal data leses inn, svar skal skrives ut og selve programmet må leses inn i hukommelsen og settes i gang. Vi husker også at det kan ta noen få millisekunder å starte én tråd, så det tar alltid noe tid før vi kan få startet parallell kjøring. Selv om vi kanskje håpet at med 1000 kjerner ville problemet vårt gå 1000 ganger så fort, så trenger vi ikke fortvile. At beregninger går fra 9 til ca. 90 ganger fortere er klart nyttige resultater.

12 OM PROSESSOREN, JAVA-KOMPILATOREN OM HUKOMMELSES-MODELLEN

I begynnelsen av kom ga vi inntrykk av at instruksjonene blir utført i den rekkefølgen de er i programteksten.

Det er ikke nødvendigvis riktig av to grunner. Først vil selve prosessoren gjerne bytte om på instruksjoner den holder på å utføre hvis det kan gå fortere, og *dersom det ikke har noen* innvirkning på sluttresultatet. Det samme prøver også java-kompilatoren. Det er mye tid å spare på å flytte rundt på instruksjoner når de utføres, f.eks. at flere andre operasjoner utføres samtidig som vi holder på med en flyttall (double)-operasjon, som tar lang tid. Slik ombytting kan bare foretas hvis *det ikke får innvirkning på sluttresultatet.* Prosessorkjernene har også nå fått egne instruksjoner som kan kjøre flere instruksjoner av type flyttall i parallell, f.eks. multiplikasjon dersom variablene til disse multiplikasjonene, som skal utføres, ligger etter hverandre i lageret. Slik omorganisering av rekkefølgen på instruksjoner må imidlertid ikke gå ut over slik vi har tenkt når vi laget programmet - programlogikken. Trenger vi resultatene av én beregning i høyresiden i en annen beregning, eller i for eksempel i en utskrift til skjerm eller fil eller i en test, blir *ikke* slik ombytting eller parallellkjøringer av instruksjoner foretatt.

Det Java og prosessoren garanterer deg er at du får utført et program som gir eksakt samme resultat som om programmet ble utført instruksjon etter instruksjon, ovenfra og nedover, en etter en, og hver gang du bruker verdien på en variabel, er den der som om programmet ditt ble utført enkelt og greit ovenfra og nedover.

Siden en programmerer nesten aldri merker effekten av slik ombytting av instruksjoner av prosessoren og kompilatoren, behøver vi ikke dvele mer med det unntatt å advare mot en feil man som programmerer kan gjøre. Se på flg. to linjer i et program:

```
x = 12;
y = 19;
```

Program 12.1*. Tilordning av verdier til to variabler. Hvis vi senere i programmet tester og finner at y er lik 19, kan vi* ***ikke*** *dermed slutte at x er lik 12 (selv om det ser ut som x=12 ble utført 'før' y=19). Siden både prosessoren og java-kompilatoren kan bytte om på tilordningen av verdier til de to variablene, og utsette 'x=12' til verdien av x enten brukes i en annen beregning, i en test eller i en utskrift, kan det godt hende at y får sin verdi lenge før x får sin verdi.*

Vi kan konkludere at den gamle enkle modellen om at vi har en prosessor og en kjerne og at den leser og utfører instruksjonene en etter en, ovenfra og nedover, ikke er helt riktig, men at det ikke gjør noe i de aller fleste tilfeller i et program med bare én tråd. Dette er viktig for oss som programmerere. Uten denne enkle modellen vil det nesten ikke være mulig å skrive riktige programmer.

12.1 HVA SKJER NÅR VI SYNKRONISERER FLERE TRÅDER PÅ SAMME OBJEKT

Når to eller flere tråder synkroniserer på **samme** objekt (sier f.eks await på samme CyclicBarrier , eller kaller en metode som er beskyttet ReentrantLock) vil alle disse oppleve følgende:

- Alle kode som er ovenfor synkroniserings-setningen i trådene og som hittil ikke er utført (f.eks, er utsatt av optimaliseringsgrunner), blir utført.
- Alle data som er skrevet på av de synkroniserende trådene blir skrevet ned i hovedhukommelsen (eller i alle fall en *felles* nivå3 cache, noe som er vanlig på en CPU med mange kjerner).

Dette betyr at alle tråder som har synkronisert på felles variabel ser samme verdier på falles variable slik de er hittil skrevet på av de deltagende trådene. Tråder som derimot ikke har synkronisert på dette felles synkroniseringsobjektet, er ikke garantert å se de siste verdiene på slike felles data.

Det er også slik at alle felles data (som har blitt endret av en av trådene) blir skrevet ned i hovedhukommelsen når trådene er ferdige, dvs. er ferdige med siste setning i run-metoden.

13 MER OM OPTIMALISERING

Noen programmer er man avhengig av at går raskt, og da oppdager man en egenskap med Java, at første gang utfører en viss type kode kan koden ta en viss tid, mens neste gang kan samme koden gå mye raskere. Kjøre man samme koden veldig mange ganger kan den typisk gå fra 20 til flere 1000 ganger raskere enn første gang. Hvorfor kan dette skje?

Det er riktig å påpeke at de fleste programmeringsspråk kan optimaliseres på denne måten slik som C, Fortan og C#. To ting bør man merke seg:

- Denne optimaliseringen kan gjøres totalt av kompilatoren når vi kompilerer vårt program og da kan man f.eks i C spesifisere om man vil ha o1, o2 eller o3 optimalisering, og da blir hele programmet optimalisert og blir da en del større enn den holdningen Java har, at bare den koden som virkelig blir brukt kompileres til maskinkode, og det er antall ganger den koden-biten blir eksekvert som avgjør hvor sterkt den etter hvert optimaliseres. I Java er det også slik at hvis det brukes en klasse eller metode fra en biblioteks-klasse (som Arrays.sort i Fig 17.3 nedenfor) må bytekoden fra biblioteket først lastes ned før den kan kompileres og så utføres, og blir da langsommere første gang den utføres enn kode som ligger i selve programmet.
- I hvor stor grad det er mulig å optimalisere kode avgjøres i stor grad om språket som skal optimaliseres er statisk typet eller ikke (dvs. at hver variabel får fastlagt sin type som int eller String, deklarert i selve koden eller ikke). De fleste vanlige programmeringsspråk er statisk typet. Men i Pyton, som ikke er statisk typet, kan en variabel 'x' for eksempel av og til inneholde et flyttall og av og til en tekst i samme programutførelse. Det er først når programmet kjører og f.eks. setningen: y = x+ 1 skal utføres er at typen til x må bestemmes, og det skjer da under selve kjøringen og forsinker utførelsen og begrenser selvsagt da også hvor mye koden kan optimaliseres.
- Uansett hvordan og hvor mye et program optimaliseres, så vil det utad, dvs. det som skrives ut til fil, på skjerm e.l. gi det samme resultat som et uoptimalisert program. Dvs. at programmet ble utført slik vi har lært i begynnerundervisningen, ovenfra og nedover, linje for linje til vi er ferdige.

Vi vet fra innføringskapitlene at først oversetter *javac* (java-kompilatoren) koden – f.eks klassen *MittProgram* din til noe som heter byte-kode (det ligger på filen *MittProgram.class).*Dette er kode for en tenkt maskin med enkle byteinstruksjoner. Så kaller du opp selve kjøresystemet *java* (som også kalles JVM – Java Virual Machine) for å kjøre programmet. Det første den gjør er å oversette all byte-kode den eksekverer til maskin-instruksjoner på den maskinen du bruker. Den oversetter ikke hele programmet ditt, bare de delene (metodene og klassene) du virkelig utfører. Første gang du kjører får du altså tider som både er oversetting til maskinkode + selve kjøretida for din kode.

Hvis den så kjøre en viss del av koden din flere ganger (f.eks 10-5000 ganger), vil den begynne å optimalisere på den maskinkoden den har laget. Enda mer kjøring kan gi ytterligere optimalisering. Optimalisering kan grovt sett beskrives som at den lager enda lurere og raskere maskin-kode av de delene av programmet du kjører ofte. Slik optimalisering kan gå i 2-3 omganger og bli stadig raskere. Det du derimot skal være sikret, uansett hvor mye maskinkoden forbedres, er at vi har sekvensiell semantikk. Dvs *du kan store på at du kan tenke og feilsøke programmet ditt ut fra selve Java-koden og du kan tenke på den som om koden blir utført ovenfra og nedover, linje for linje, løkke for løkke som om det aldri ble utført noen oversettelse til maskinkode eller senere optimalisering av denne.*

La oss se på et eksempel:

```
class C { int i;
   C(int i){ this.i =i;}
   int les () { return i+10;}
} // end C

s =”new Class C + metodekall “;

t = System.nanoTime();
for (int i = 0; i<n; i++) {
       k = new C(k).les();
}
d = (double) ((System.nanoTime() -t)/(n*1000.0));
println(s+d + «us.snitt « + n+» ganger»);
```

Vi ser at dette er en kode som først gjør en *new C(k)* og så kaller les-metoden i objektet vi har laget av klassen C og skriver ut tiden for det. Så gjør den det samme, bare n ganger. Første gang tok det 2697 µs (milliondels sekund), mens det med n=2 tok i snitt 0.45µs. En forbedring på ca 5000 ganger raskere! Selvsagt kan vi gjøre samme for ulike Java-konstruksjoner (se tabellen nedenfor). Grunnen til at det går mye raskere er ikke bare at vi oversetter til maskinkode første gang, men også at det i JVM bygges opp datastrukturer for dine klasser og metoder slik at neste gang har JVM en mye lettere og raskere jobb med å kjøre ditt program.

n, ant. ganger	1	2	3	100	10000	100000	X bedre, (speedup)
for-løkke	0,3	0,15	0,03	0,018	0,009	0,007	**42**
metode-kall	2697	0,45	0,06	0,054	0,026	0,026	**103730**
new int[100]	1,2	0,6	0,24	0,195	0,151	0,136	**33**
array copy med for-loop, n=100	1,8	1,5	2,64	2,500	1,177	0,188	**9**
System.arraycopy, n=100	5,7	0,3	0,15	0,126	0,072	0,064	**89**
new Thread med start & join()	3015	336	66,6	61,68	61,87	61,86	**48**
new C(int) og metodekall	2697	0,45	0,15	0,21	0,035	0,035	**77 057**
Int [] array read	0,3	0,3	0,06	0,036	0,012	0,012	**25**
Innstikk-sortering (n=100)	46,6	42,8	42,42	21,27	19,60	1,45	**32**

Tabell 12.1 Kjøretider fra sekvensielt testprogram i **2017** i µs med Java 1.8.0 for ulike Java-konstruksjoner

og et enkelt sorteringsprogram som funksjon av antall utførte ganger + speedup for n = 100 000, Intel i7-7600 @3,4 Ghz.

n	1	2	3	100	10000	1000000	x bedre (SU)
for-loop	0,7	0,2	0,023	0,023	0,002	0	**350,0**
metodekall	9,7	0,9	0,1	0,102	0,005	0	**1940,0**
int[] new	0,6	0,3	0,152	0,24	0,137	0,087	**6,9**
copy array for-loop	2	2,3	2,793	1,336	0,635	0,011	**181,8**
arraycopy	4	0,7	0,208	0,036	0,033	0,021	**190,5**
new Thread,start&join	2576,9	301,8	210,659	175,962	0		**14,6**
new Class C m.metodekall	2301,6	8,4	0,272	0,011	0	0	**209236,4**
int [] a skriv	0,6	0,5	0,028	0,006	0,01	0,007	**60,0**
int [] les	0,3	0,2	0,023	0,005	0,013	0,007	**23,1**
double to long	4,2	1,2	0,164	0,007	0,044	0	**95,5**
insertSort int[]	118,3	154	47,332	3,201	1,65	1,771	**71,7**
Arrays.sort	471	70,9	45,023	4,68	1,221	1,187	**385,7**

Tabell 12.2. En Litt forenklet versjon av fig .2.1. Kjøretider fra **2022** i µs + speedup fra n=2 til n= 1000 000, Java 14.0 for ulike Java-konstruksjoner og to sorteringsprogram som funksjon av antall utførte ganger av et sekvensielt testprogram på en AMD Ryzen 5 3500U, 2.0 3.4GHz. Kommentar: New Thread eksemplet er bare kjørt for n= 1,2,3 og 100, pga tidsforbruket.

Tallene ovenfor varierer noe for hver gang de kjøres, men viser klare trekk som at særlig metodekall og det å lage objekter effektiviseres ekstremt, men at arrayer ikke effektiviseres i samme grad. Oppmuntrende er det at brukerkode som en innstikks-sorterings algoritme som vi har skrevet (og som hver gang sorterer en ny usortert double array av lengde 100) kan effektiviseres med en faktor ca. 70 .

Delvis kommer det at din kode blir forbedret, men også at de data og variable har om ditt program blir kraftig forbedret. Uansett behøver en Java-programmerer bry seg om hvordan det skjer, men bare vite at ethvert Java-program med tiden vil gå stadig fortere, men alltid produsere samme svaret.

Dette at optimaliseringen gir oss samme svaret som om vi ikke hadde optimalisert koden, kalles ***sekveniell konsistens*** og er meget viktig for oss når vi skal feilrette programmet. Vi kan tenke på vårt program som om det blir utført linje for linje, ovenfra og nedover. Selve optimalisingen kan man **slå av** med å starte programmet vårt slik etter kompileringen til bytekode:

```
>java -Xint MittProgram  … de vanlige parametrene …
```

(brukes bare ved debugging hvis man tror at optimaliseringen har 'ødelagt' programmet, noe den nesten aldri har gjort.)

Det er viktig å vite at kallet på en metode eller det å lage et objekt av en klasse er knyttet til ***kallstedet***. Det betyr at hvis du f.eks kaller samme metode fra et annet sted i din kode, vil den også bli optimalisert der på ny. Det som grovt sett skjer er at det ikke blir foretatt et hopp til en optimalisert metode, men at denne optimaliserte koden blir lagt inn direkte der kallet på metoden er. Vi kan si at kallet som blir optimaliser ved å bli fjernet på kallstedet og erstattet med metodens kode.

Likevel kan av dette lære oss en del om hvordan vi bør skrive våre programmer:

1. Siden metodekall og det å bruke klasser og lage objekter av disse effektiviseres sterkt, bør vi ikke være redde for å bruke disse i rikt monn i våre programmer. Mange algoritmer kan være ganske kompliserte med mange steg og løkker. Hvis hvert slik steg kan utformes som

en egen, mindre metode, og at det hele så bygges sammen med en overordnet metode som i rekkefølge kaller disse mindre metodene, får vi et program som både er lettere å forstå og debugge, og mulig også raskere (optimalisatoren liker godt små metoder).

2. Når vi skal lage parallelle programmer med tråder, ser vi at det bare er det første trådobjektet som tar lang tid å lage og starte/avslutte (koster ca. 2,5 millisek). De neste, nr 2,3,.. tar hver bare ca 1/10-del av denne tida. Antall tråder vi deklarerer i et parallelt program behøver derfor ikke ha mye å si for kjøretiden. Ofte vil f.eks. det å bruke flere tråder som vi har kjerner være et vellykket valg for løsning av et parallelt problem.
3. Ofte er det også slik at noen problemer, som sortering av tall, går så raskt at det ca. 4 millisek. å sortere si 25 000 tall med en sekvensiell metode. Det gjør at en sekvensiell sorteringsalgoritme vi være ferdig før den parallelle versjonen av algoritmen har greid å ha fått starte sine tråder og begynt å løse problemet, De fleste parallelle programmer vil derfor inneholde en første metode av typen:

```
void løsProblemet ( ..param..) {
    if(størrelsen_av_problemet   <   minimum_størrlse)
        løsProblemetSekvensielt(..param..);
    else løsProblemetParallelt)..param…);
}// end løsProblement
```

4. Med få unntak, ikke prøv å optimaliser koden selv, optimalisatoren er langt flinkere enn deg.
5. Unntakene er at arrayer med to dimensjoner alltid bør leses, beregnes og skrives ***radvis*** og sekvensielt, og at de data vi leser/skriver i en løkke helst bør passe inn i størrelsen på cache, nivå1 eller cache-nivå 2.
6. Du skal selvsagt, når du lager en parallellisert løsning på et problem, bruke den raskeste, sekvensielle metoden som et utgangspunkt for den parallelle løsningen.
7. Husk at når vi lager en parallell løsning med *k* tråder har vi laget *k* sekvensielle programmer som deler felles kode, men som har ulike deler av problemets data. Hvis de data som deles har blitt delt opp i *k* deler, er det å forvente at hver del passer langt bedre inn i cache nivå 1 og 2. Hvis det er tilfellet, vil et slikt parallelt program kunne gå raskere enn *k* ganger fortere enn det opprinnelige sekvensielle løsningen fordi mer av data ligger høyere opp i cache-hierarkiet.
8. Derimot er det et annet forhold som peker mot at programmet ikke går så fort, og det er forbindelsen mellom kjernene og hovedhukommelsen, kalt datakanalene. Når *k* kjerner samtidig vil skrive og lese via datakanalene i hovedhukommelsen, blir det ofte kø og ventetider med langsommere eksekvering som resultat.
9. For å oppsummere, det er et rimelig håp at et parallellisert program på en multikjerne PC med *k* kjerner, vil ha en speedup på ca. *k.*

13. Feilaktige antagelser i PRAM-modellen

Som tidligere nevnt har det blitt laget en teori for å analysere og lage parallelle programmer som kalles PRAM (Parallell Random Access Machine). Den gjør følgende forenklinger for lettere å analysere parallelle maskiner og programmer:

1. Tiden det tar å lese eller skrive i hukommelsen er konstant og den samme, og settes =1.
2. Vi kan lage programmer som kan ha så mange kjerner vi vil - f.eks. *n* kjerner hvis vi skal sortere *n* tall.
3. Alle parallelle programmer kan startes samtidig på samme klokke-sykel
4. De parallelle kjernene går synkront, dvs. hvis de utfører samme program, vil de i én klokke-sykel utføre eksakt samme instruksjon på samme tidspunkt.

Alle disse påstandene er ganske langt fra virkeligheten og gale, og kan gi feilaktige analyser og dysfunksjonelle programmer fordi:

1. Påstanden om konstant tid for lesing og skriving ignorerer prefetch og cache-systemet. Leser/skriver vi data som befinner seg i et register eller i cache nivå1 vet vi at det går 100-200 ganger fortere enn om vi får en cache-feil og vi må da gå helt ned i hovedhukommelsen for data til hver ny instruksjon. I praksis opplever vi tidsforskjeller på minst faktor 20-40 hvis en litt stor (si minst 100x100) todimensjonal array aksesseres kolonnevis istedenfor radvis.

2. Feilen med at vi kan ikke fritt skrive programmer med veldig mange tråder (si mer enn ca. 1000-100 000) og i tillegg tro at dette går greit fordi man i parallell har like mange kjerner. Dette vil det raskt fylle hele data-maskinen, da hvert objekt av klassen Thread tar en viss plass. Og siden ingen multikjerne maskin i praksis har mer enn 32-64 kjerner, vil det bli en evig køing av tråder på å få eksekvere på en virkelig kjerne med mye utskifting av systemdata i hovedhukommelsen og kø på data-kanalene. Antagelsen av vi har valgfritt antall kjerner tilgjengelig vil lett få oss til å skrive dysfunksjonelle programmer.
   Riktignok kan vi med å programmere GPU-en (grafikkprosessoren) kunne få flere tusen prosessorer i parallell som utfører samme instruksjon på ulike data. Men her er vi begrenset til noen tusen kjerner, og er det bare noen typer problemer som løses effektivt med grafikkprosessoren. Dette kurset et kurs i å programmere parallelle algoritmer på en multikjerne CPU. Tilleggskurs i GPU- og programmering av beregningsklynger samt snart også på kvantemaskiner finnes på Ifi. Generelt kan det sies at ikke alle typer av problemer lar seg løse mer effektivt på slike spesielle maskiner enn på en multikjerne PC.

3. og 4 : Antagelse om full synkronisering er gal. I praksis startes trådene så raskt man greier sekvensielt av et sekvensielt program. Hvis trådene f.eks. skal gjøre veldig lite (si øke en felles variabel 100 ganger), så kan vi lett oppleve at vi ikke får noen feil fordi den første tråden som starter er ferdig før neste tråd får begynt på å utføre sin kode. Vi kan da lett få det inntrykket at vi ikke trenger å synkronisere adgangen til denne felles variabelen med f.eks en *ReentrantLock* eller *synchronised.* Elektronikken i kjernene er heller ikke alltid synkronisert slik at vi fysisk sett kan få litt ulik hastighet på de ulike kjernene.

Riktignok har PRAM-modellen blitt modifisert i ulike retninger. og vel særlig antagensen om all aksess til data tar samme tid, men kurset IN3030 finner det uproduktivt og galt å basere sin programmering på en sterkt forenklet teori og modell for parallellprogrammering. Vi trenger ingen annen modell annet enn programmet for problemene vi skal løse. Å se bort fra en rekke mekanismer i elektronikken og i optimaliseringen i kjøresystemet java (JVM Java Virtuell Machine), finner vi uproduktivt og tidvis villedende for å finne den beste løsningen og algoritmen for et problem.

Kurset IN3030/4040 baserer seg på tidtaking som målestokk på en effektiv algoritme i tillegg til at vi nytter O(n) - analyse av både den sekvensielle og parallelle algoritmen for å fine forventet kjøretid som funksjon av antall dataelementer n.

14 MENGDENE I JAVA KAN VÆRE LANGSOMME

Når man ser på hastighet i Java-programmer, så bør man også se på at i alle mengdene i Java-biblioteket som var mengder av basale typer som int, double, long ... byttet ut med tilsvarende klasse- representasjonen av tilsvarenede basale typer som Integer for int, Double for double, …osv. Vi deklarerer en ArrayList for heltall slik, og her legger vi samtidig inn tallene 1 til n i lista:

```
ArrayList<Integer> liste1 = new ArrayList <Integer> ();
for (int i = 1 ; i <= n; i++) liste1.add(i);
```

Det som skjer bak kulissene er at hver int-verdi blir pakket inn i et Integer objekt før det blir lagret med en peker fra lista til dette objektet. (på engelsk: boxing) . Skal vi så lese verdien i Integer-objektet , må heltallsverdiene pakkes ut byte for byte (på engelsk: unboxing) fra et Integer-objekt på 16 byte + 8 byte til en peker til objektet i motsetning til en 'vanlig' int i en array på 4 byte . Det vil si at en Integer i en array tar omlag 6x så stor plass som en int-variabel, noe som gjør at de raskere fyller opp de ulike cachene, og det tar lenger tid for å gjøre les og skriv på heltallselementene man tross alt behandler.

For å vurdere hvor rask ArrayList er, kan man lage en alternativ implementasjon *IntList* med en heltallsarray med den funkjonalitetet som ArrayList har:

```
class IntList{
    // representerer k heltall, adressert: 0..k-1
    int [] data;
    int len =0;
    IntList(int len){data = new int [Math.max(1,len)];}
    IntList() {data = new int [16]];}

    void add(int elem) {
    if (len == data.length) {
        int [] b = new int [data.length*2];
        System.arraycopy(data,0, b,0,data.length);
        data =b;
          }
          data[len++] = elem;
      } // end add

    void clear(){len =0;}

    int get (int pos){
        // error: antar at svaret brukes til array-indeks
        if (pos > len-1 ) return -1; else return data [pos];
    } // end get

    int size() {return len;}
} // end class IntList
```

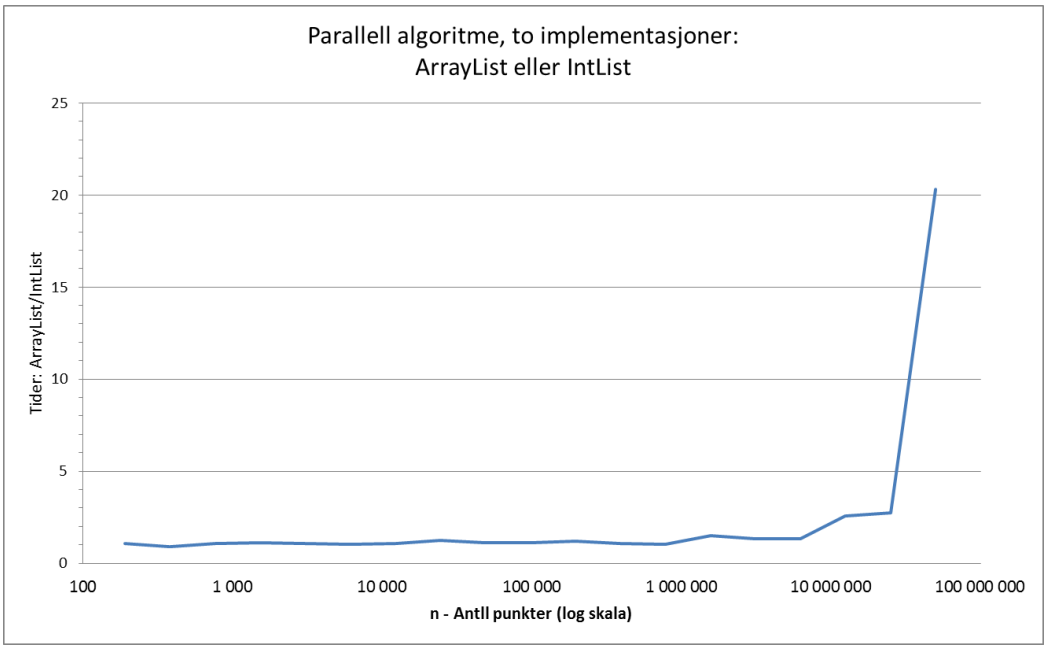
Kjører vi en enkel test av innlegging (add) og les(get) n ganger ser vi at ArrayList er klart langsommere enn IntList .

Tabell 13.2 *Forholdet i kjøretider for ArrayList/IntList for ulike lengder av listene, og viser hvor mange ganger langsommere ArrayList er enn IntList.*

n	IntList(ms)	ArrayList(ms)	forhold
500 000 000	6 194.84	211 277.47	34.11
50 000 000	146.46	2 054.59	14.03
5 000 000	15.06	41.22	3.05
500 000	1.13	3.44	3.43
50 000	0.10	0.36	1.57
5 000	0.02	0.04	2.50
500	0.00	0.00	2.00

Grunnen til at det ikke blir verre forsinkelse er at ArrayList, som mange av de andre mengdene, er parallellisert (for 'store' verdier av n). Hvis du bruker ArrayList i en parallell tråd, kan ny parallellisering være en dårlig ide, fordi istedenfor *k* parallelle tråder har du plutselig *k*k* parallelle tråder som slåss om de *k* fysiske kjernene. Dette gir vanligvis hastighetsreduksjon. Parallelle tråder bør ikke selv starte sine *k* tråder osv.

Kjører vi et parallelt program som bruker IntList eller ArrayList mye (testet på et som løste den konvekse innhyllinga til n punkter), får om lag like store forsinkelser ved å bruke ArrayList. Derfor bruker vi IntList.



Figur 17.*1 Vi ser at jevnt over er den parallelle algoritmen som bruker IntList ca 10-15% raskere når n er liten, men blir når n > 10 mill blir IntList mye raskere enn ArrayList.*

15 OPPSUMMERING – OM PARALLELLISERING AV ET PROBLEM

Det er mange måter å parallellisere et problem i en multikjerne-maskin, men de to som er forklart her, bruk av synkroniserte metoder, barriere synkronisering samt Reentrantlock, skulle være tilstrekkelig for de aller fleste problemer. Særlig barrieresynkronisering egner seg for større problemer.

1. For at det skal være vits å parallellisere et problem, må det ha *mer* enn noen få operasjoner for hvert dataelement. Som en huskeregel kan vi si at hvis den største versjonen av problemet vi greier å kjøre sekvensielt ikke tar mer enn 0.01 sekund, er det ingen vits i å parallellisere det.
2. Start med en godt testet og effektiv **sekvensiell** løsning av problemet.
3. Se etter om man kan dele ***data*** opp i et antall like store deler, hvor helst hver del kan løses etter den sekvensielle metoden i hver sin tråd – altså i parallell.
4. Hvis/når vi under beregningene må ha felles data, må de alltid beskyttes. All skriving og lesing på disse felles data skjer med synkroniserte metoder.
5. Hvis ***alle data alltid*** er felles, er det ingen vits i å parallellisere problemet hvis ikke trådene hver kan ha kopier av relevante felles data og at disse til sist greit kan samstilles etter beregningene.
6. Vi kan godt uten beskyttelse la ulike tråder i parallell skrive på **ulike elementer** i en array (men ***ikke*** i ulike bit i samme byte)
7. Når hver tråd har løst sin del, og etter at alle har ventet på *en felles barriere* når de er ferdige, så kan alle trådene etterpå lese resultatene av hverandres beregninger.
8. Kanskje er vi nå enten ferdig, eller resultatene fra første beregninger kan igjen deles opp i flere tråder med en ny barriere., osv.
9. Tenk spesielt på hvordan main-tråden skal vente og slippe løs når alle trådene er ferdige og har løst problemet. Det kan godt være en barriere som venter på antall tråder +1, som er main-tråden, som main legger seg og venter på når alle trådene er startet.
10. For noen klasser av problemer lønner det seg å sette i gang flere tråder enn man har *k* kjerner (inkludert hyperthreaded kjerner) i maskinen – f.eks 2k,3k eller 4k tråder. Bare test om dette evt. går fortere. Grunnen til en hastighetsøkning her er at de deler av data vi behandler i en tråd passer bedre i cache-systemet. Ulempene ved å få oftere byte av kjerner og mer synkronisering oppveies da av hastigheten av cache-systemet.

To typiske parallelliseringer med *k* kjerner og *k* tråder:

A. De sentrale data i problemet lar seg enkelt dele i *k* like deler hvor vi kan la den sekvensielle algoritmen løse hver sin 1/*k* del av problemet med hver sin tråd i parallell Hvis problemet er så enkelt at man skal finne det størst elementet i en array, vil vi til sist etter at hver tråd har køet på en CyclicBarrier, så kan en av trådene (f.eks. tråd nr. 0) sammenligne de *k* svarene og rapportere den største av disse på skjermen eller fil.

B. Hvis den sekvensielle løsningen er rekursiv, er det en enkel men ikke helt optimal løsning å erstatte de få øverste rekursive kallene med at vi lager en tråd og for hver av disse. Det er viktig at der er toppen av rekursjonstreet (de øverste 2 til 4 lagene hvor de rekursive kallene er erstattet av en parallell tråd). Problemet med denne løsningen er at den første oppdelingen i toppen med to rekursive kall blir sekvensiell. Det er først på neste lag at vi kan få til 2-parallell, så 4-parallell på neste nivå osv. Det er for noen rekursive algoritmer som Kvikksort mulig å endre disse sli at vi kan starte med full parallell med alle trådene.[https://ojs.bibsys.no/index.php/NIK/article/view/247] . For andre rekursive algoritmer som Flettesortering mulig å heve parallelliteten til 2-parallell , så 4-parallell,…, på det øverste laget i rekursiviteten ved at de kortere sorterte delene flettes både fra starten for å finne de minste elementene og samtidig fra endene sorteres for finne de største elementene i parallell [https://ojs.bibsys.no/index.php/NIK/article/view/491]

OPPGAVER

1. Lag den sekvensielle versjonen av Kvikksort, modifiser utførOgTest() for sorteringseksempelet slik at du lager en array med samme innhold som har blitt sortert av sQuick og sorter den med Arrays.sort(). Skriv ut tiden for denne og sammenlign med tiden for kvikksorttiden.
2. I den parallelle versjonen av Kvikksort, pQuick, fjern de rekursive kallene og løs problemet bare med tråder. Kommenter kjøretidene og antall tråder du får. Er dette lurt?
3. Skriv et program for 'En Selgers rundtur'; først sekvensielt (rekursiv) og så en blanding av parallelt og rekursivt. Dette er en oppgave med svært mange beregninger og svært lite data, og egner seg svært godt for parallellisering.
   Problemet er slik: En selger skal reise og besøke n byer – hver by skal besøkes bare en gang. Hun vet avstandene fra enhver by til alle de andre byene. Disse dataene har hun i en todimensjonal array: **`avstand[][]`**, slik at avstand [i][j] er avstanden fra by i til by j. Om avstandene vet du også at **`avstand[i][j]= avstand[j][i]`**, at **`avstand[i][i]`** = **0**, og at det alltid er raskest å reise direkte til en by – det går aldri noen snarvei ved å reise via en annen by.
   Skriv ut den korteste reiseplan av alle mulige hvor selgeren besøker hver by bare en gang og som kommer til slutt tilbake til utgangsbyen.

```
class SeqRSelger {
  int [][] avstand ;
  int [] x,y;
  int bestHittil =Integer.MAX_VALUE;
  int [] besteRute, reiseRute;
  boolean [] besøkt;
  int n;
  void  RSR (int nivå, int by, int lengde) {
      if( nivå == n) { // gjenstår bare reisen tilbake til by:0
           if (lengde+avstand[by][0] < bestHittil){
                 <notér ny beste reisevei og lengde>
           }
      } else {
          for (int nesteBy = 1; nesteBy < n ; nesteBy++) {
            // prøv å besøke alle ikke-besøkte byer unntatt 0
            if (! besøkt [nesteBy]) {
               besøkt[nesteBy] = true;
               reiseRute[nivå] = nesteBy;
               RSR (nivå+1,nesteBy,lengde +avstand[by][nesteBy]);
               besøkt[nesteBy] = false;
   }}}} // end PSR

   SeqRSelger(int n) {
         <opprett og initier arrrayer med tildeldige tall>
         <beregn avstandsmatrisen med Pytagoras>
    } // end konstruktor
    void utfør() {
         < ta starttidspunkt>
          RSR(1,0,0);
          <skriv ut data, tid brukt på PSR() og beste reisevei>;
    }
     public static void main (String [] args) {
          new SeqRSelger(Integer.parseInt(args[0])).utfør();
     }
} //end SeqRSelger
```

Programskisse til oppgave 3. *Selgerens rundreise. Dette er den sekvensielle varianten av problemet. Få denne til å virke og lag så en parallell versjon. Dette er på ingen måte den beste algoritmen som løser dette problemet, men nok den korteste. Merk hvor raskt den løser et 10-bys problem, men at det tar alt for lang tid å løse et 15-byes problem. Kjøretiden går som* ***n!*** *(=1*2*..*(n-1)*n). Vi ser at kjøretiden da øker meget raskt, eksponensielt med n.*

4. Lag en versjon av parallell Kvikksort hvor du bare bruker én sykliske barriere. Hint: Ser du på kall-treet i fig. 1.7 ser du at det er ett kall på **pQuick** på nivå 1 (toppen) i treet, samlet 3 kall på nivå 2 og nivå1 i treet, samlet 7 kall til og med nivå 3 osv. (formelen for antall kall med k nivåer er: 2^k-1). Du må da før du starter kallene regne ut hvor mange nivåer du vil ha i kall-treet for å starte nye tråder, og sette opp den sykliske barrieren til å vente på så mange. Resten av problemet løser du som før rekursivt. Er denne raskere enn den som står i avsnitt 8.1?

REFERANSER PÅ NYERE PROSESSORER

1 -Amazone Graviton : https://www.nextplatform.com/2022/01/04/inside-amazons-graviton3-arm-server-processor/

https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd

2: Intel :
https://images.anandtech.com/doci/17596/VIP%20Acceleration%20Experience%20Primer%20Meeting%20Slides%208.png

3: AMD EPYC: HTTPS://EN.WIKIPEDIA.ORG/WIKI/TEMPLATE:AMD_EPYC_7003_SERIES

4: Apple : https://www.apple.com/newsroom/2022/03/apple-unveils-m1-ultra-the-worlds-most-powerful-chip-for-a-personal-computer/

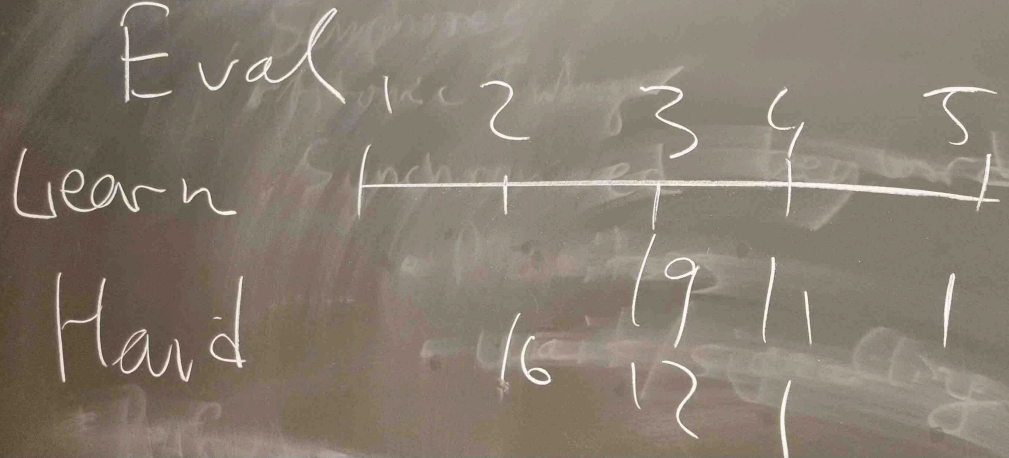
Amdahl's Law

Gustarson's Law

IN 4330

JMH

cache misses



Multicore Architectures

- Moore's Law
- Basic layout.
- Speedup, def
- Amdahl's law
- Caching

Threads in Java

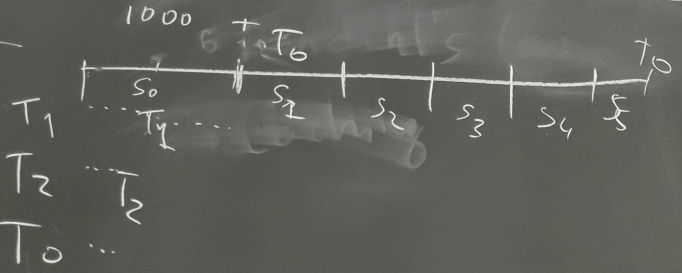
- how to create
- performance
  - create
  - GC

Caching

- slow speed of light.
- ① Latency
  - Latency hiding technique
- ② Tradeoff / size vs speed
- why Multi level Caches

Timing

- "warm up"
- Why Median/Minimum.



Parallelization

- Split data
- Split work
- Divide & Conquer
- Iterative
- Recursive

Synchronization

Semaphore sem = new Semaphore(1);

Critical Section

On entry : sem.acquire();

At exit sem.release();

t1 sem.acquire()

t2 sem.acquire()

sem.release(); sem.release();

Want a shared counter

int k = 0;

inc() - increase by one

get() - returns the current value of the counter

Barrier: 3
Threads: 4

AAABAB

Barrier 2
Threads 3

AABABB

AABBB

IVar

```
int k
Semaphore ready = new Semaphore(0)

int get()
    int r;
    ready.acquire()
    r = k
    ready.release()
    return r

void put(value)
    k = value
    ready.release()
```

```
int initialized = 0;
Semaphore sem = Semaphore(1)

put(value)
    sem.acquire()
    if (initialized)
        sem.release()
        return False
    else initialized = 1
        k = value
        ready.release
        sem.release
        return True
```

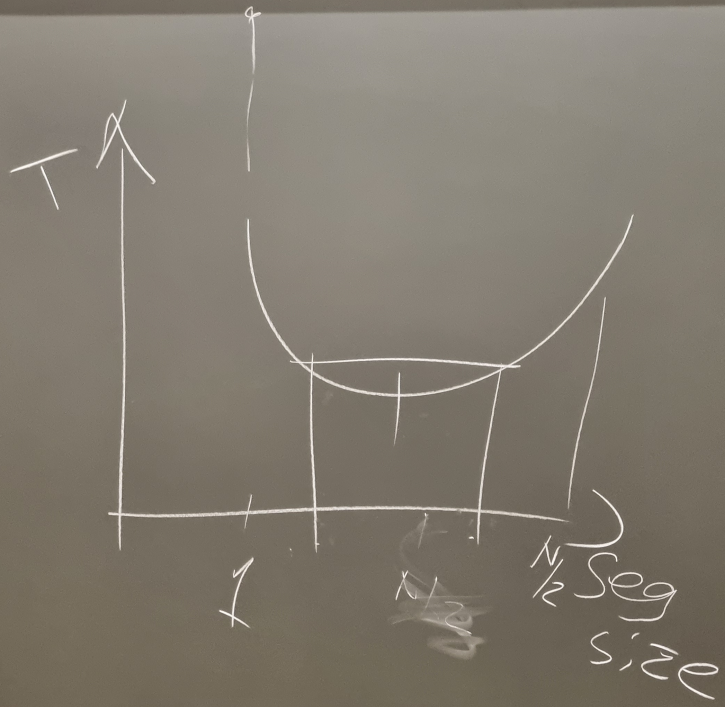
```
int k=0; Semaphore sem = new Semaphore(1);

void inc() {
    sem.acquire()
    k = k+1
    sem.release()
}

int get() {
    sem.acquire()
    return k
    sem.release()
}
```

```
int get()
    int r;
    sem.acquire()
    r = k;
    sem.release()
} return r
```

```
sem.acquire()
int i = get()
print i
sem.release
```

Kitchen

```
Semaphore sem = new Semaphore(2)

Enter kitchen
    sem.acquire

Leave
    sem.release
```


T
1
N/2
Seg
size

k

1.

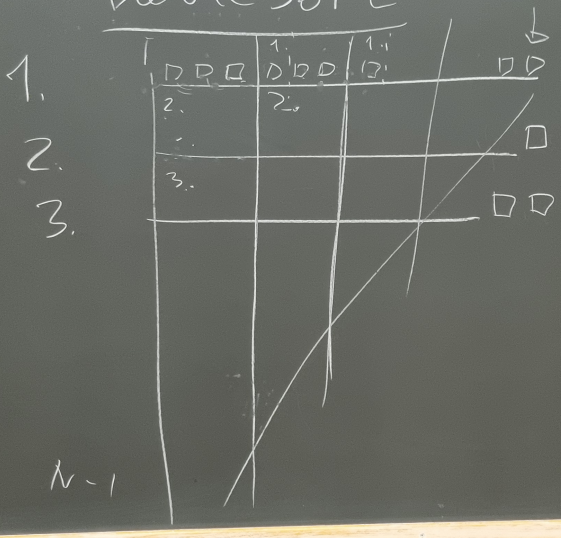
2.

3.

⋮

1. $k+1$

Beblesort



$N-1$

$N-2$

$N-3$

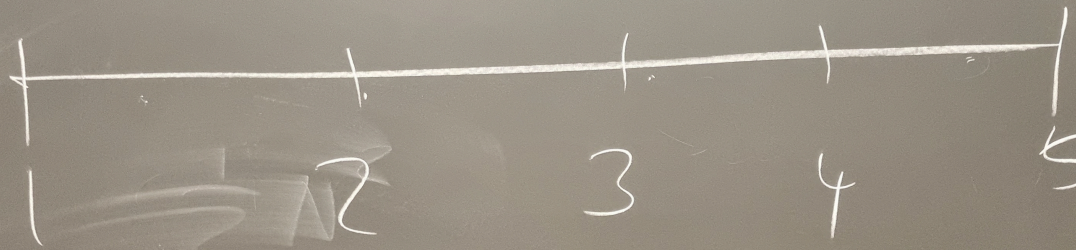
$\frac{1}{2}N^2$

L17

- REMAING EXERCISES
- EXAM 31/5
- HINT FOR EXAM
- EXAMPLES
- Q&A
- EVAL

EVAL ENTIRE COURSE

LEARN 7 29 2



HARD

19 11 1

Oblig 1 in IN3030/IN4330 – v2023
Find the *k* largest numbers in a large array

February 1st, 2023

One problem for web search programs such as Google, Bing and DuckDuckGo is that a search can generate millions or even billions of answers (if you searched for "football" in Google using Chrome today, I got 3.49 billion answers!) – more or less relevant to the one who requested the search. She / he will never be able to look at all the answers, but will merely like to look at the most relevant.
Each page (of the many billions of hits) has a relevance score, which we can assume is an integer so that the the larger this number is, the more relevant the page is. The problem then is to select the most relevant answers.

Let us help DuckDuckGo parallelize the solution to this problem. Suppose you have n answers and that the relevance score is stored in the integer array a[0..n-1]. A simple sequential solution A1 in Java is to use Javas built-in sort algorithm (Arrays.sort (int [] a)) to sort the entire array and then pick out the *k* largest numbers as a[0..k-1]. But this takes way too long because the sorting is typically $\mathcal{O}(n \cdot \log n)$.

A faster algorithm A2 is the following:

1. We first note that the first *k* numbers in a[0..k-1] are, obviously, the largest of the first k numbers in a[]. We then insert-sort a[0..k-1] in descending order (you must trivially rewrite the normal insert-sort (given below) to sort in descending order – *i.e.*, with the largest first).
2. Then we know that the smallest number of the first *k* numbers in a[] is in a[*k-1*]. Then we compare a[*k-1*] in turn with each element in the rest of the array a[*k*..n-1]. If we find an element a[j] where a [j]> a [*k-1*], then we do the following:
   a. Replace a [*k-1*] with a[j].
   b. Insert-sort the new element into a[0 .. *k-2*] in descending order (remember that the code to sorting only one item is easier than full insertion sorting).
3. When step 2 is done, the *k* largest numbers are in [0.. *k-1*] and none of the other numbers has been overwritten or broken.

Task 1 - Sequential Algorithm

Implement the sequential algorithm A2 above. Test it for these different values of n = 1000, 10,000, …, 100 million by creating an array of pseudo-random numbers (java.util.random) and for each of these values, you test for two values of *k* = 20 and *k* = 100. Furthermore, test that you get the correct answer by sorting the same numbers Arrays.sort (int [] a) and compare your answers (descending order) with the corresponding *k* places in a[] after you have use Arrays.sort () to check that it is correct.
(Note to get the same 'random' numbers in the array when the Random class if you want to redo the run several times, the constructor of the Random class must get a starting number that is the same as previous runs – *e.g.*, **Random r = new Random (7363);** Then we will get the same number sequence when we say: **r.nextInt (n)** in the loop that gets the next number between 0 and n-1**).** The numbers n and *k* are included as parameters when you start your program, *e.g.,* as:

```
java myprog <n> <k>
```

Write two tables, one for *k* = 20, one for 100, showing the time the two different methods A1: Arrays.sort and A2: Insert method uses for different values of n (n = 1000, 10,000, ..., 100 million). You should also produce two curves, one for k = 20 and one for k = 100 showing the results. The times reported must be the median of 7 calls on both methods as shown in the lecture week2. Submit your code to A2 and the table with your comments.

Optionally, you are welcome to explain why A2 is faster than A1 – even to give the asymptotic behavior of A1 and A2.

Task 2 - Parallel Algorithm

You should parallelize A2 as best you can with the *p* cores you have on your machine. Use an IfI machine, if necessary to have at least 4 cores. Take, for example, the starting point the parallelization of the FinnMax problem. You may then be left with a small sequential phase after that most of the calculations are done (as in FinnMax).

Submission

Write a report with two tables, one for $k = 20$ and one for $k = 100$, showing the times that Arrays.sort (), sequential and parallel insertion method A2 uses for different values of n (n = 1,000, 10,000, …, 100 million).
In addition, a graph with two curves is also preferred, one for $k = 20$ and one for $k = 100$, showing speedup as a function of n. The times reported here should be the median of 7 calls on both methods as shown in one of the lectures. Submit your code to both the sequential and parallel solution and the report.

The report should include what type of CPU (name and speed in GHz and the number of cores that it has, and, if available, the make and model of the CPU chip) you used; as well as comments on what value of n you observe that the parallel code gives speedup > 1, or why it does not achieve a speedup > 1 for that value of *k*. Also comment on how the execution times change for the two choices of *k.*

Submissions in IN3030/IN4330 are done thru Devilry.

Oblig1 must be done individually and submitted no later than

Tuesday February 14th 2023 at 23:59:00 (*NOT* 23:59:**59**)

NOTE: you can submit *multiple* times – only the last submission will be considered, so we **ENCOURAGE** you to submit one or more versions early so that you are sure that you do *not* miss the deadline – the deadline is **HARD!**

Tips

1) You might get an error message when you try to run your program for n = 100 million saying that you have too little memory. If so, you can start the program with an option for increased memory. To request 6 GB for the program use:

```
java myprog -Xmx6000m 100000000 <+ other parameters>
```

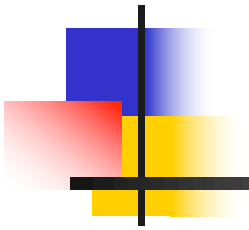
If you do not have 64-bit Java, this does not work, max is then 1000m that you can ask for. If necessary, use a machine at IfI – or download 64-bit Java 8 to your machine from:
http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Appendix: Code for insert-sort a[0.. *k-1*];

a) Note: This sort is in **ascending** order – you have to rewrite it yourself and have it sorted in **descending order**:

```
/** This sorts a [v..h] in ascending order with the insertion algorithm */
void insertSort (int [] a, int v, int h) {
  int i, t;
  for (int k = v; k < h; k++) {
    // invariant: a [v..k] is now sorted ascending (smallest first)
    t = a[k + 1];
    i = k;
    while (i> = v && a[i]> t) {
      a[i + 1] = a[i];
      i--;
    }
    a[i + 1] = t;
  } // than for k
} // end insertSort
```

b) Note: To insert a new item at place a[*k-1*] you need a simpler version of the code above because the first *k* -1 elements in a[0 .. *k-2*] have already been sorted into descending order.



IN3030/IN4330
Effektiv parallellprogrammering
L1, (Uke 4), våren 2023

Eric Jul
Professor
Gruppeleder Programmeringsteknologi
Institutt for Informatikk
Universitetet i Oslo
Norge

Recording of Lectures

- **Recording lectures: I will attempt BUT I make no promises – a few times last spring my recording did not work ☹**
- **Lecture slides are mostly in Norwegian (perhaps a little Danish intermixed in a few places…)**

Do not be shy...

- **Ask any question at ANY TIME during the lecture – I shall try to respond fairly quickly ☺**
- **Also, REMEMBER, in education:**

THERE ARE NO STUDPID QUESTIONS, only STUPID ANSWERS!

English

- **I am Danish: so my Danwegian sounds like a terrible dialect of Norwegian – spoken only about 200 km southeast of Kristianssand.**
- **Many Norwegians understand what I say, but, alas, many that have Norwegian as a second language have trouble doing so**
- **And there are a few international students in the course.**
- **So I will lecture in English.**

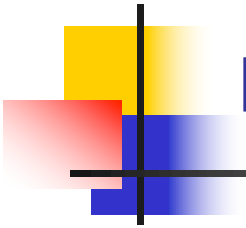
Hvem

- **Meg**
- **Arne Maus –** original course designer, Emeritus
- **Michael Kirkedal Thomsen –** new førsteamanuensis in PT ☺

- **2 Teaching Assistants**
- **3 «Rettere»**

- **About 150 students signed up (usually somewhat fewer show up)**

Hvorfor dette kurs?



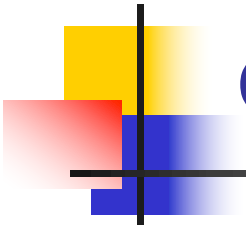
- **Love of *Learning by Doing***
- **Oppleve progammer, der gjør en forskjell!!**
- **Uttnytte *multicore – multikjerne***
- ***Teori er bra, MEN praksis essentiel!***
- ***Bli bedre programmør!!***
- ***… And have fun doing so!!!***

Litt om Eric

- **Ph.D. University of Washington, 1989**
- **Dansk-amerikaner**
- **Bor i Danmark – pendler til Oslo ca 2-3 gange/måned**
- **1989-2000: Førsteamauensis Københavns Universitet**
- **2000-2009: Professor Københavns Universitet**
- **2009-2015: Bell Labs, Dublin & Professor II, IfI**
- **2016- Professor IfI**
- **2019- Gruppeleder *Programming Technology***

- ***Favorite hobbies: Skiing and Gliding (Seilfly) and a litte bicycling***

Litt mere om Eric



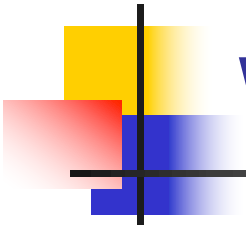
Communication

- **Please keep up with the messages on the web site: The course website is THE source of information**
- **Use the Hjeplelærer ☺**
- **Ask questions when in doubt – we will be setting up a Q&A Forum**
- **IN4330 students – use the IN3030 web site**

Bakgrunn IN3030

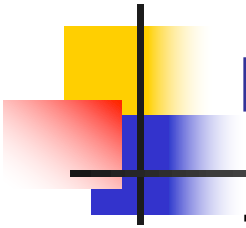
- **Kurs er fra 2014 – tidligere INF2440**
- **Laget av Arne Maus – siden 2018 Emeritus**
- **Overtaket av Eric 2018**
- **I 2019 endring: INF2440 -> IN3030/IN4330**

- **IN4330: versjonen for Masters student: bruk BARE IN3030 web!**



Why are YOU here??

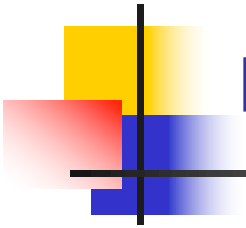
- **What do you expect from the course?**



Let's get started on parallelism

Two examples:

- **Plant a tree**
  - **Embarrassingly paralizable!**
- **Make a baby**
  - **Inherently sequential!**



Motivation for Parallelle Programmer

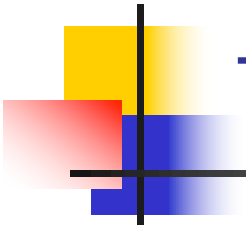
- **Maskiner kan bestå av flere CPU-er**
  - Hver CPU kan utføre programmer uavhengig av de andre CPUer
- **Hver CPU kan have flere kjerner**
  - Hver kjerne kan utføre programmer
- Vi vil gjerne utnytte disse muligheter for parallellisme

Hva vi skal lære om i dette kurset:

Lage parallelle programmer (algoritmer) som er:

- **Riktige**
  - Parallelle programmer er klart vanskeligere å lage enn sekvensielle løsninger på et problem.
- **Effektive**
  - dvs. raskere enn en sekvensiell løsning på samme problem

- Lære hvordan man parallelliserer et riktig, sekvensielt program + lage egne parallelle algoritmer som ikke bare er en slik parallellisering;
- Lære de mange problemene vi støter på og hvordan disse kan takles.
- Kurset er **empirisk** (med tidsmålinger), ikke basert på en teoretisk *modell* av parallelle beregninger,
- Vi oppfatter programmet som en god nok modell av det problemet vi skal løse. Vi trenger ingen modell av modellen.
- Presenterer en klassifikasjon av parallelle algoritmer (nytt).



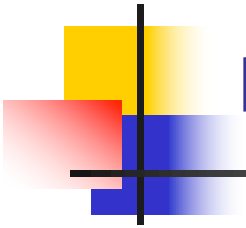
Tre grunner til å lage parallelle programmer

1) Skille ut aktiviteter som går **langsommere** i en egen tråd.
   - Eks: Tegne grafikk på skjermen, lese i databasen, sende melding på nettet. Asynkron kommunikasjon.
2) Av og til er det **lettere** å programmere løsningen som flere parallelle tråder. Naturlig oppdeling.
   - Eks: Kundesystem over nettet hvor hver bruker får en tråd.
   - Hele operativsystemet har mye parallellitet – 1572 aktive tråder i Windows 7 akkurat da denne foilen blev skrevet i 2017 – og 1921 aktive tråder i Mac OS X Sierra.
3) Vi ønsker **raskere** programmer, raskere algoritmer.
   - Eks: Tekniske beregninger, søking og sortering.

Dette kurset legger nesten all vekt på raskere algoritmer

IN3030 - et **nytt** kurs – og dog

- IN3030 er 90% baseret på INF2440.
- Et relativt UNIKT kurs – set internationalt.
- Planlagt fem obliger - ca. 2-3 uker per oblig.
  - De individuelle innleveringer : Man kan samarbeide om algoritmer, men **ikke** ha helt lik eller delvis felles kode med andre. Rapport skal skrives selv.
- En oblig er ikke bare innlevering av ett eller flere parallelle programmer, men **også en liten rapport** om de testene man har gjort: hastighetsmålinger på disse for ulike størrelse av data med konklusjoner – f.eks speedup + O( ) og en forklaring på resultatene .
- Gruppetimer:
  - Jobbing med ukeoppgaver og obligene
- Kurs under utvikling betyr at også dere selv vil være med på forme kurset, og at ikke alt vil være perfeekt.



Pensum

- Ingen dekkende lærebok er funnet, men:
  - **Kompendium av Arne Maus:** legges opp på web
  - **Det som foreleses (Powerpoint slides) + oppgavene (obliger + ukeoppgaver) er pensum.**

  - **Bra bok:** Brian Goetz, T.Perlis, J. Bloch, J. Bobeer, D. Holms og Doug Lea::"Java Concurrency in practice", Addison Wesley 2006
  - Kap. **18 og 19** i A. Brunland, K. Hegna, O.C. Lingjærde, A. Maus:"Rett på Java" 3.utg. Universitetsforlaget, 2011.
  - I tillegg leses *fra en maskin på Ifi* kap 1 til 1.4, hele 2 og 3.1 til 3.7 (hopp over programeksemplene) i :

http://www.sciencedirect.com/science/book/9780124159938

(Hvis leses utenfor Ifi, så koster det $!)

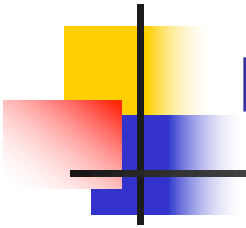
I dag – teori og praksis

- Ulike maskiner og kurs – hvor plasserer INF2440 seg?
- Begrunnelse for multikjerne CPU og parallelle løsninger/algoritmer.
- Parallelle løsninger på et problem er lengere (ofte minst dobbelt så lang kode) og (en god del) vanskeligere å lage enn en sekvensielt algoritme som løser samme problem.
- Den eneste grunnen til å lage parallelle algoritmer er at de går fortere enn samme sekvensielle algoritme – i alle fall for tilstrekkelig stor n (= antall data).
- Vi måler hvor-mange-ganger-fortere-det-går – speedup S:

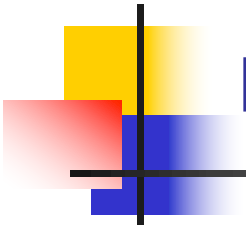
$$S = \frac{tid\ (sekvensiell\ algoritme)}{tid\ (parallell\ algoritme)}$$

- som da skal være > 1 , men vi skal også lage og teste programmer som har S < 1 og forklare hvorfor.

Lineær speedup ?



- Selvsagt ønsker vi lineær speedup – dvs. bruker vi *k* kjerner skulle det helst gå *k* ganger fortere enn med 1 kjerne.
- Meget sjelden at det kan oppnås (mer om det siden)
- Kan superlineær speedup oppnås?
  - *PLEASE THINK ABOUT THIS!*
- Speedup: a central metric.

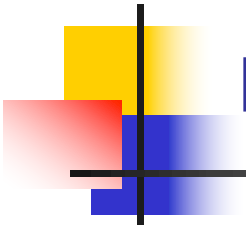


Lineær speedup analogier

- Selvsagt ønsker vi lineær speedup, MEN:
- Noen problem er **nemme** at parallellisere: Eksempel: 10 personer kan plante 10 treer ca 10 gange fortere end 1 person kan plante 10 treer.
- Andre problemer er *vanskeligere* at parallellisere: Eksempel: hvis 1 person kan samle et Lego-set på 10 timer, så vil det være vanskelig for 10 personer at samle det på 1 time – der er avhengigheter mellom delene.
- Andre problemer er nærmest ***umulig*** at parallellisere: Eksempel: hvis 1 kvinne kan lage et barn på 9 måneder, kan 9 kvinner så lage et barn på 1 måned?

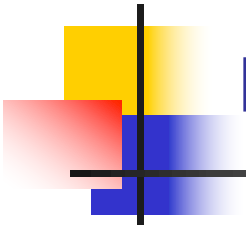
Flynns klassifikasjon av datamaskiner:

	Single Instruction	Multiple Instruction
Single Data	SISD : Enkeltkjerne CPU	MISD : Pipeline utførelse av instruksjoner i en CPU. Flere maskiner som av sikkerhetsgrunner utfører samme instruksjoner.
Multiple Data	SIMD : **GPU** – samme operasjon på mange elementer (en vektor) Det finnes også slike SSE – instruksjoner på Intel og AMDs CPU-er	**MIMD :** klynge av datamaskiner, og **Multikjerne CPU**



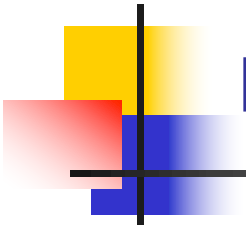
Dette kurset handler om multikjernemaskiner

- Nå har vi multikjerne maskiner!
  - Din 2500 NOK bærbar: 2-4 kjerner
  - Min 2017 MacBook Pro: 8 kjerner
  - Stor server: 64-128 kjerner
- Hvordan uttnytter vi dem??
- Svar: vi bruker tråde!



Dette kurset handler om tråder og effektivitet

- Hva er tråder
  - Se litt på maskinen
  - Se på operativsystemet
  - Hvordan skal vi oppfatte en tråd i et Java-program
  - Senere se på kompileringen og kjøring av Java-kode
- Hvordan måle effektivitet
  - Hvordan ta tiden på ulike deler av et program; både:
    - Den sekvensielle algoritmen
    - En eller flere parallelle løsninger
  - I dag: Enkel tidtaking
  - Neste gang: Bedre tidtaking
- Praktisk i dag
  - Standard måte å starte programmet med tråder

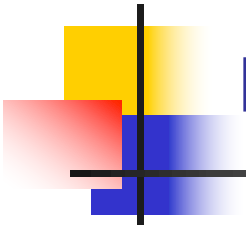


Flere mulige synsvinkler

Mange nivåer i parallellprogrammering:

1. Maskinvare
2. Programmeringsspråk
3. Programmeringsabstraksjon.
4. Hvilke typer problem egner seg for parallelle løsninger?
5. Empiriske eller formelle metoder for parallelle beregninger

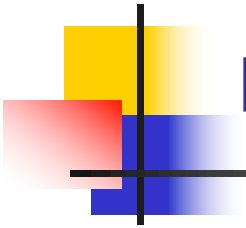
IN3030/IN4330: Parallellprogrammering av ulike algoritmer med tråder på multikjerne CPU i Java – empirisk vurdert ved tidsmålinger.



Learning-by-doing

Dette kurs er et Learning-by-doing kurs!

Utbyttet ditt er avhengig av **DIN** innsats!



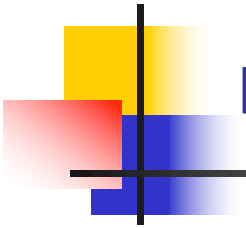
DIN innsats

Dette kurs er et Learning-by-doing kurs!

Go program lots of parallel programs

AND

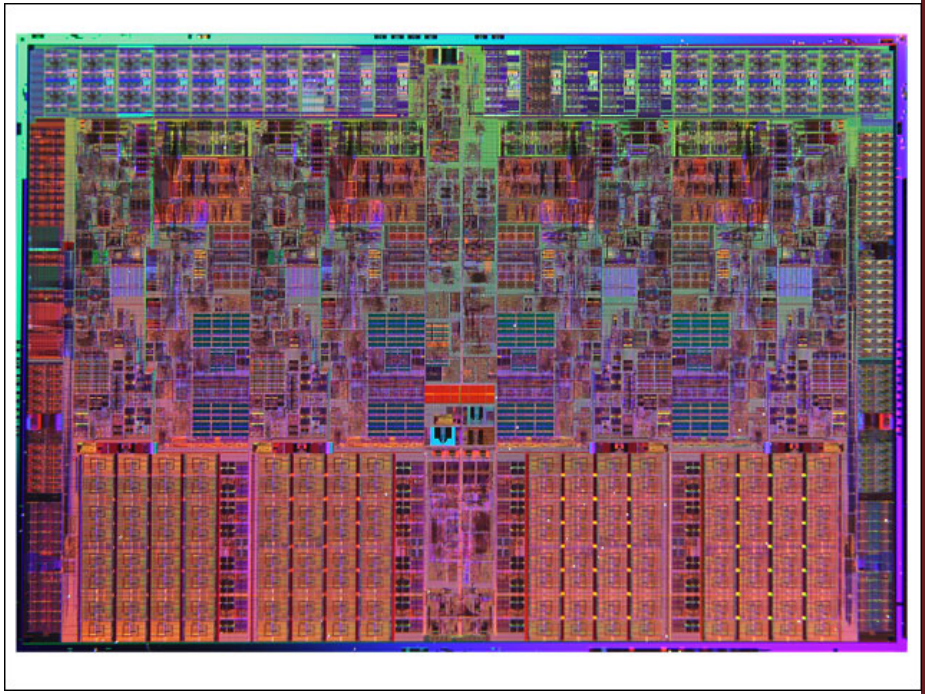
make them ***FAST***

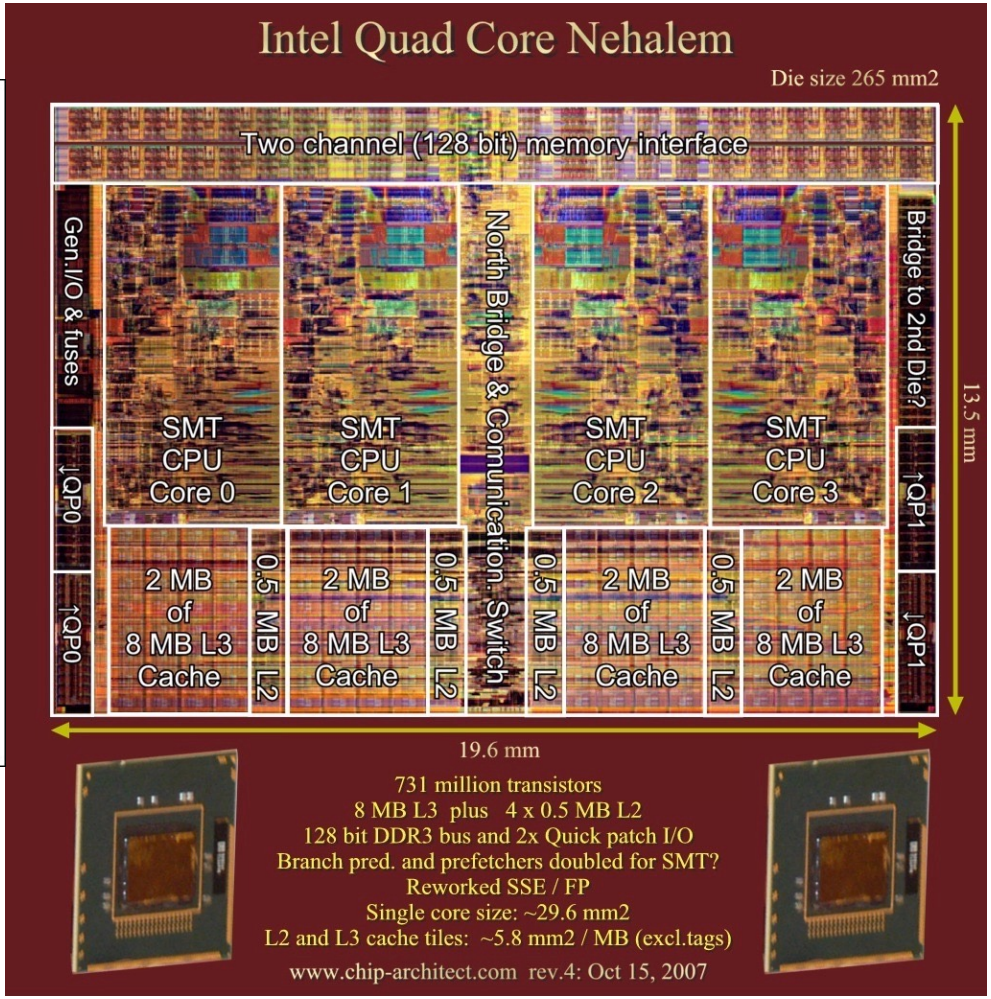


Maskinvare og språk for parallelle beregninger og Ifi-kurs

1. Grafikkkort GPU med 2000+ små-kjerner, **IN5050**
   - Eks Nvidia med flere tusen småkjerner (SIMD – maskin)
2. Mange maskiner løst koblet over internett. **IN5570**
   - Planetlab – world-wide testbed
   - Emerald – språk til distribuert programmering
3. Mange maskiner løst koblet i utkanten av nettet. **IN5600**
   - Fog Computing
4. Teoretiske modeller for beregningene **IN5170**
   - PRAM modellen og formelle modeller (f.eks FSM)

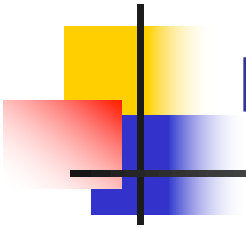
Multikjerne - Intel Multicore Nehalam CPU





Mange ulike deler i en Multicore CPU – bla. en pipeline av maskininstruksjoner; kjernene holder på med 10 til 20 instruksjoner ***samtidig*** dvs. instruksjonsparallellitet

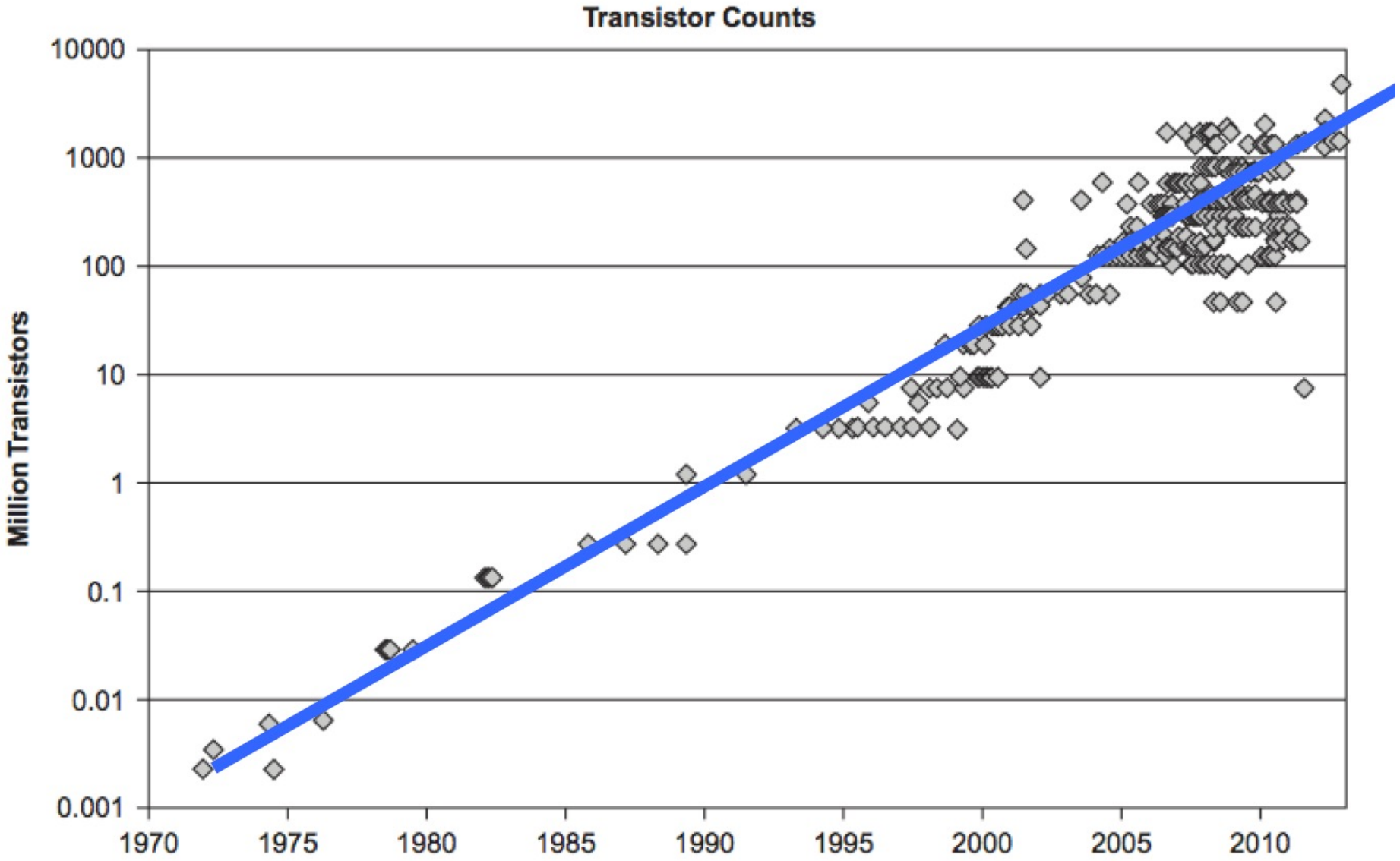
Hvorfor får vi multikjerne CPUer ?

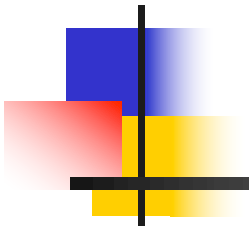


- Hver 18-24 måned dobler antall transistorer vi kan få på en brikke: Moores lov
- Vi kan ikke lage raskere kretser fordi da vil vi ikke greie å luftkjøle dem (ca. 120 Watt på ca. 2x2 cm – varmere enn en rødglødende kokeplate). Og der er kvantemekaniske begrensninger også.
- Med f.eks dobbelt så mange transistorer ønsker vi oss egentlig en dobbelt så rask maskin, men det vi får er dessverre 'bare' dobbelt så mange CPU-kjerner.
- Med flere regnekverner (kjerner) må vi få opp hastigheten ved å lage parallelle programmer !

Transistors per Processor over Time

Continues to grow exponentially (Moore's Law)

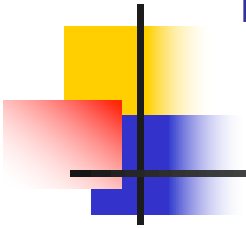




IN3030/IN4330 – Effektiv parallellprogrammering Lecture L02, våren 2023

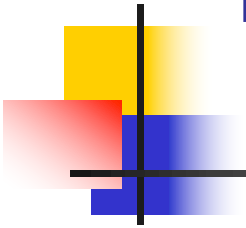
Eric Jul
Professor
PT
Institutt for Informatikk

Resume of first lecture v2023



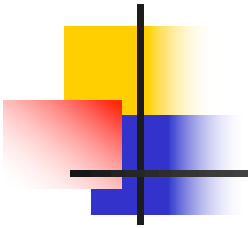
- Motivation
  - Utilization of Multi-core – by changing sequential programs into parallel programs
  - Multi-core hardware driving this trend
- Purpose
  - Convert sequential program into parallel versions
  - Achieve speedup
- Requirements
  - Correctness – Effective!
  - Efficient – MUST be FASTER – measured by speedup
- Approach
  - Empirical – *the proof of the pudding is in the eating!*
  - *Timings have the ULTIMATIVE SAY!*
  - *(Theory is fine; but practise is essential!)*

Resume of first lecture v2021 (second page)



- Central metric: SPEEDUP
  - Speedup: sequential time / parallel time
  - Want speedup > 1
  - Really want speedup = *number of cores*
- How?
  - Parallel threads in Java
  - Must synchronize
- Evaluation?
  - Real-time clock times!!
- Multi-core architecture
- Non-uniform memory access
  - Multi-level caching
- Threads in Java

SO: how do we utilize the parallel power of the hardware??



- **For now: Multithreading!**
- Plant a hundred trees ☺ Using a hundred persons.
- Create the concept of a thread
  - One (somewhat non-elegant) version is in Java
- Problem with concurrent updates
- Synchronization required – otherwise loss of data and incorrect programs
- Synchronization can lead to significant source of overhead!!

Tråder i Java

- Er én programflyt, dvs. en serie med instruksjoner som oppfører seg som ett vanlig, sekvensielt program – og kjører på én kjerne
- Det kan godt være (langt) flere tråder enn det er kjerner.
- En tråd er ofte implementert i form av en indre klasse i den klassen som løser problemet vårt (da får trådene greit aksess til **felles data**):

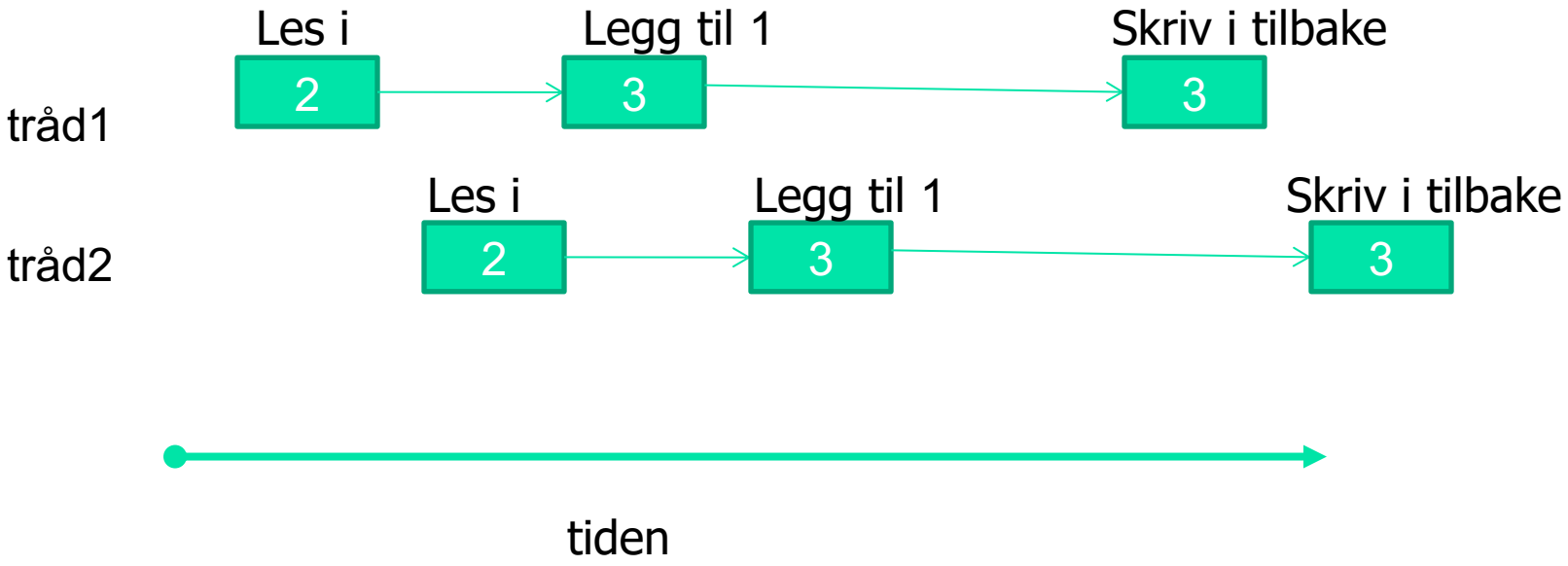
```
import java.util.concurrent.*;
class Problem { int [] fellesData ; // dette er felles, delte data for alle trådene
        public static void main(String [] args) {
                Problem p = new Problem();
                        p.utfoer();
                }
                void utfoer () { Thread  t = new Thread(new Arbeider());
                    t.start();
                 }

            class Arbeider implements Runnable {
                int i,lokalData; // dette er lokale data for hver tråd
                    public void run() {
                            // denne kalles når tråden er startet
                    }
             } // end indre klasse Arbeider
} // end class Problem
```

Example: Concurrent update of a variable Let's try! (Spoiler: we will fail!)

```
import java.util.concurrent.*;
class Problem { int i ; // dette er felles, delte data for alle trådene
        public static void main(String [] args) {
                Problem p = new Problem();
                        p.utfoer();
                }
                void utfoer () {
                    int j, k;
                    j=10;
                    for (k=0; k< j; k++)  {
                        new Thread(new Arbeider()).start();
                 }

          class Arbeider implements Runnable {
               int i,lokalData; // dette er lokale data for hver tråd
                  public void run() { // denne kalles når tråden er startet
                        i++;
                  }
           } // end indre klasse Arbeider
} // end class Problem
```

1) Ett problem i dag: operasjoner blandes ved samtidige oppdateringer

- Samtidig oppdatering - flere tråder sier gjentatte ganger: **i++ ;** der i er en felles int.
  - **i++** er 3 operasjoner: a) les i, b) legg til 1, c) skriv i tilbake
  - Anta i =2, og to tråder gjør i++
  - Vi kan få svaret 3 eller 4 (skulle fått 4!)
  - Dette skjer i praksis !



Test på i++; parallell

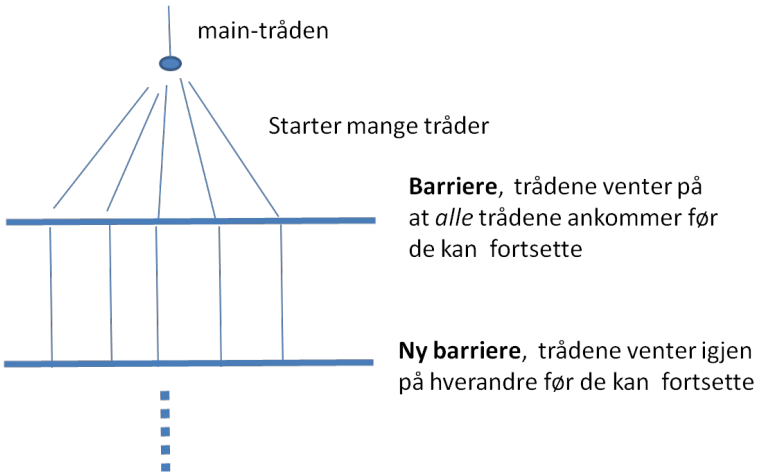
- Setter i gang **n tråder** (på en 2-kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger uten synkronisering;

```
for (int j =0; j< 100000; j++) {
        i++;
}
```

- Vi fikk følgende feil - antall og %, (manglende verdier).
  Merk: Resultatene *varierer også mye* mellom hver kjøring :

Antall tråder n		**1**	**2**	**20**	**200**	**2000**
Svar	1.gang	100 000	200000	1290279	16940111	170127199
	2.gang	100 000	159234	1706068	16459210	164954894
Tap	1.gang	0 %	0%	35,5%	15,3%	14,9%
	2. gang	0%	20,4%	14,6%	17,7%	17,5%

Kommende program bruker CyclicBarrier. Hva gjør den?



- Man lager først ett, felles objekt `b` av klassen CyclicBarrier med et tall: `ant` til konstruktøren = det antall tråder den skal køe opp før alle trådene slippes fri 'samtidig'.
- Tråder (også main-tråden) som vil køe opp på en CyclicBarrier sier await() på den.
- De `ant-1` første trådene som sier await(), blir lagt i en kø.
- Når tråd nummer `ant` sier await() på `b`, blir alle trådene sluppet ut av køen 'samtidig' og fortsetter i sin kode.
- Det sykliske barriere objektet `b` er da med en gang klar til å være kø for nye, `ant` stk. tråder.

Praktisk: skal nå se på programmet som laget tabellen

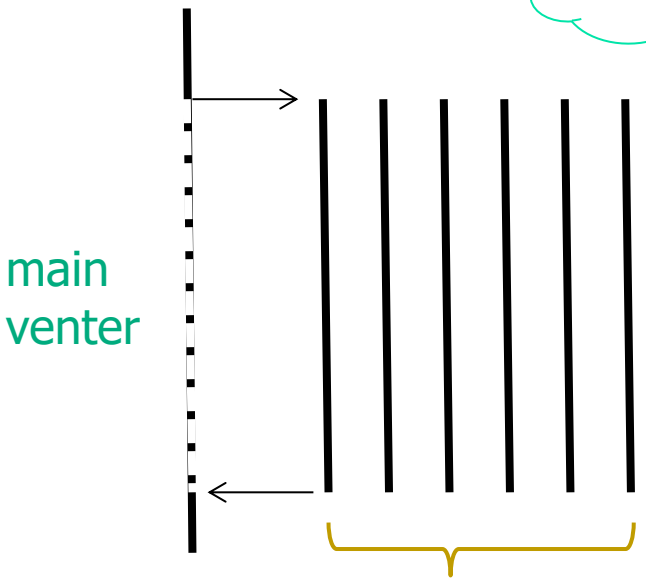
```
import java.util.*;
import easyIO.*;
import java.util.concurrent.*;
/** Viser at manglende synkronisering på ett felles objekt gir feil – bare loesning 1) er riktig'*/

public class Parallell {
        int tall;                               // Sum av at 'antTraader' traader teller opp denne
        CyclicBarrier b ;                       // sikrer at alle er ferdige naar vi tar tid og sum
        int antTraader, antGanger ,svar; // Etter summering: riktig svar er:antTraader*antGanger

        // det kommer I alt 4 forsøk på å øke i, bare en av dem er riktig
        //synchronized void inkrTall(){ tall++;}       // 1) –OK fordi synkroniserer på ett objekt (p)
        void inkrTall() { tall++;}                      // 2) - feil

        public static void main (String [] args) {
                if (args.length < 2) {
                        System.out.println("bruk >java Parallell <antTraader> <n= antGanger>");
                }else{  int antKjerner = Runtime.getRuntime().availableProcessors();
                        System.out.println("Maskinen har "+ antKjerner + " prosessorkjerner.");
                         Parallell p =  new Parallell();
                          p.antTraader = Integer.parseInt(args[0]);
                         p.antGanger = Integer.parseInt(args[1]);
                         p.utfor();
        } } // end main
```

```
void utskrift (double tid) {
        svar = antGanger*antTraader;
        System.out.println("Tid "+antGanger+" kall * "+ antTraader+" Traader ="+
                                                    Format.align(tid,9,1)+ " millisek,");
         System.out.println(" sum:"+ tall +", tap:"+ (svar -tall)+" = "+
          Format.align( ((svar - tall)*100.0 /svar),12,6)+"%");

} // end utskrift

 void  utfor () {   b = new CyclicBarrier(antTraader+1);       //+1, også main
                    long t = System.nanoTime();                 // start klokke

                    for (int j = 0; j< antTraader; j++) {
                          new Thread(new Para(j)).start();
                     }

                   try{   // main thread venter
                          b.await();
                    } catch (Exception e) {return;}
                    double tid = (System.nanoTime()-t)/1000000.0;
                    utskrift(tid);
   } // utfor
```

```
class Para implements Runnable{
        int ind;
        Para(int ind) { this.ind =ind;}

        public void run() {
            for (int j = 0; j< antGanger; j++) {
                inkrTall();
              }
              try {  // wait on all other threads + main
                  b.await();
              } catch (Exception e) {return;}
          } // end run

        //    void inkrTall() { tall++;}                    // 3) Feil - usynkronisert
        //    synchronized void inkrTall(){ tall++;}  // 4) Feil – kallene synkroniserer på
                                                                //      hvert sitt objekt
    } // end class Para
} // END class Parallell
```

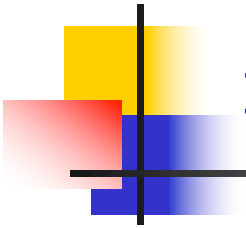
Husk: Vanligste oppsett av main-tråden + k tråder

main, lager k nye tråder



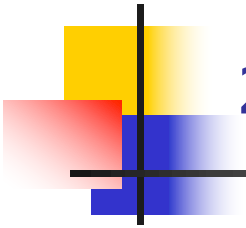
k tråder, leser og skriver i egne og i felles data og løser problemet

Hver av trådene (main + k nye) er sekvensielle programmer.
Problemet er at de samtidig *ikke kan skrive* på felles data



1) Avslutning med en CyclicBarrier

- En CyclicBarrier (cb= new CyclicBarrier (n+1))
  - Er tenkt som et ventested, en bom/grind for et antall (i dette tilfellet for n+1) tråder - de n 'nye' trådene + main. Alle må vente når de sier sier cb.await() til sistemann ankommer køen, og **da** kan alle fortsette.
  - Trådene kan da være ferdige med en beregning kan selv avslutte med å bli ferdige med sin run() -kode. Main-tråden forsetter, og vet at de andre trådene er ferdige. Main-tråden kan da bruke resultatene fra trådene.
  - Den sykliske barrieren cb er da strakt klar til å køe nye n tråder som sier cb.await() , .. osv
  - cb.await() sies inne i en try-catch blokk



2) Avslutning med en Semaphore

- En Semaphore (sf = new Semaphore(-n+1))
  - Administrerer (i dette tilfellet) –n+1 stk. **tillatelser**.
  - To sentrale primitiver:
    - sf.acquire() – ber om **en** tillatelse. Antall tillatelser i sf blir da 1 mindre hvis antallet er >0. Hvis det ikke er noen ledig tillatelse, må tråden vente i en kø (inne i en try-catch blokk)
    - sf.release() – gir **én** tillatelse tilbake til semaforen sf. Ikke try-catch blokk (Den tillatelsen som gis tilbake behøver ikke vært 'fått' ved hjelp av aquire() ; den er bare et tall).
  - Avlutning med Semaphore sf:
    - Maintråden sier sf.acquire() – og må vente på at det er minst en tillatelse i sf.
    - Alle de n nye trådene sier sf.release() når de terminerer, og når den siste sier sf.release() blir det 1 tillatelse ledig og main fortsetter.
    - Ikke syklisk.

3) Avslutning med join() - enklest

- Logikken er her at i den rutinen hvor alle trådene lages, legges de også inn i en array. Main-tråden legger seg til å vente på den tråden som den har peker til skal terminere selv.
  Venter på alle trådene etter tur at de terminerer:

```
 // main –tråden i konstruktøren
Thread [] t = new Thread[n];
for (int i = 0; i < n; i++) {
   t[i] = new Thread (new Arbeider(..));
   t[i].start();
 }
  ……….
  // main vil vente her til trådene er ferdige
for(int i = 0; i < n; i++) {
       try{ t[i].join();
        }catch (Exception e){return;};
}     ……..……..
```

II) Mange ulike synkroniserings primitiver
Vi skal bare lære noen få !

- java.util.concurrent

Classes

AbstractExecutorService
ArrayBlockingQueue
ConcurrentHashMap
ConcurrentLinkedDeque
ConcurrentLinkedQueue
ConcurrentSkipListMap
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet
CountDownLatch
CyclicBarrier
DelayQueue
Exchanger
ExecutorCompletionService
ecutor
eadPoolExecutor
ThreadPoolExecutor.AbortPolicy
ThreadPoolExecutor.CallerRunsPolicy
ThreadPoolExecutor.DiscardOldestPolicy
ThreadPoolExecutor.DiscardPolicy

Semaphore
SynchronousQueue
ThreadLocalRandom
ThrExecutors
ForkJoinPool
ForkJoinTask
ForkJoinWorkerThread
FutureTask
LinkedBlockingDeque
LinkedBlockingQueue
LinkedTransferQueue
Phaser
PriorityBlockingQueue
RecursiveAction
RecursiveTask
ScheduledThreadPoolEx

Interfaces

BlockingDeque
BlockingQueue
Callable
CompletionService
ConcurrentMap
ConcurrentNavigableMap
Delayed
Executor
ExecutorService
ForkJoinPool.ForkJoinWorkerThreadFacto
ry
ForkJoinPool.ManagedBlocker
Future
RejectedExecutionHandler
RunnableFuture
RunnableScheduledFuture
ScheduledExecutorService
ScheduledFuture
ThreadFactory
TransferQueue

java.util.concurrent.atomic

De har samme virkning (semantikk) som volatile variable (forklares senere), men kan gjøre mer sammensatte operasjoner. Mye raskere enn sychronized methods.

Classes

AtomicBoolean
AtomicInteger
AtomicIntegerArray
AtomicIntegerFieldUpdater
AtomicLong
AtomicLongArray
AtomicLongFieldUpdater
AtomicMarkableReference
AtomicReference
AtomicReferenceArray
AtomicReferenceFieldUpdater
AtomicStampedReference

Eksempel på operasjoner i **AtomicIntegerArray:**

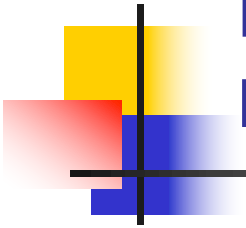
int	**get**(int i) Gets the current value at position i.
int	**getAndAdd**(int i, int delta) Atomically adds the given value to the element at index i.
int	**getAndDecrement**(int i) Atomically decrements by one the element at index
void	**set**(int i, int newValue) Sets the element at position i to the given value.

Vi skal bare lære ett fåtall av dette

- Her er de vi skal konsentrere oss om:
  - new Thread – join()
  - synchronized method
  - Semaphore – aquire() og release()
  - CyclicBarrier – await()
  - ExecutorService pool = Executors.newFixedThreadPool(k);
    med Futures - forklares senere
  - AtomicIntegerArray – get(), set(), getAndAdd(),..
  - ReentrantLock (i pakken: **java.util.concurrent.locks**)
  - volatile variable - forklares senere

- Alle de synkroniseringer vi trenger, kan gjøres med disse!
- De fleste andre har sine måter å gjøre det på, men man har neppe tid til å lære seg alle.
- Bedre å bli flink i et lite og tilstrekkelig sett av synkroniseringsprimitiver, enn halvgod i de fleste.

Kan det gå galt når to tråder samtidig skriver i ulike plasser i en array?



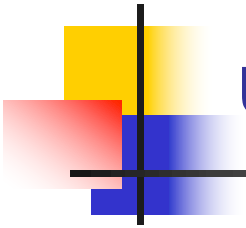
- Et problemet kunne være at når en av tråden lester opp et element i a[i] (int = 4 byte), så er cache-linja 64 byte, så den får med seg flere elementer før og etter a[i].
- Disse 'andre' elementene er det andre tråder som skriver på.
- Vi skriver et testprogram (ParaArray) hvor 10 tråder med indeks : 0,1,2,..,9 som øker hvert sitt element i en array tall[index] 100 000 ganger.

Skriving på nærliggende elementer i en array.

```
class ParaArray{
                    int  []tall;
                    CyclicBarrier b ;
                    int antTraader, antGanger ;
....
class Para implements Runnable{
    int indeks;
   Para(int i) { indeks =i;}
       public void run() {
                    for (int j = 0; j< antGanger; j++) {
                            oekTall(indeks);
                     }
                    try {  // wait on all other threads + main
                                    b.await();
                    } catch (Exception e) {return;}
       } // end run
        void oekTall(int i) { tall[i]++; }
  }
} // end ParaArray
```

- Cache-linja er nå 64 byte (og en int er 4 byte)
- Går det greit med at flere tråder (indeks=0,1,…,k-1) skriver på a[tråd.indeks] mange ganger i parallell?
- Tester: Vi lageret program som gjør det :

```
>java ParaArray 10 100000000
Maskinen har 8 prosessorkjerner.
Tid 100000000 kall * 10 Traader =
0.032600 sek,
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
```



Ukeoppgave L2

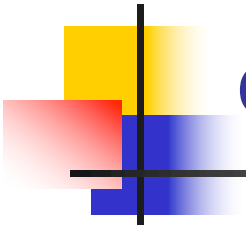
- Ukeoppgave L2 presenteres

Oblig plan

IN3030/IN4330 Oblig plan 2023:

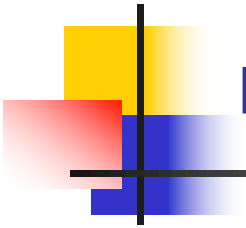
O1	L2	2 weeks	1/2 – 14/2
O2	L4	2 weeks	15/2 – 28/2
O3	L7	4 weeks+1 week of Easter	8/3 – 11/4
O4	L11	3 weeks	12/4 – 2/5

Written exam 31/5 – 2023

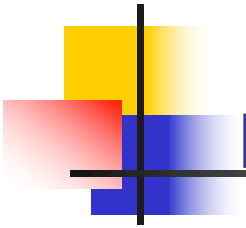


Oblig 1

- Oblig 1 presenteres



End of lecture L2v23

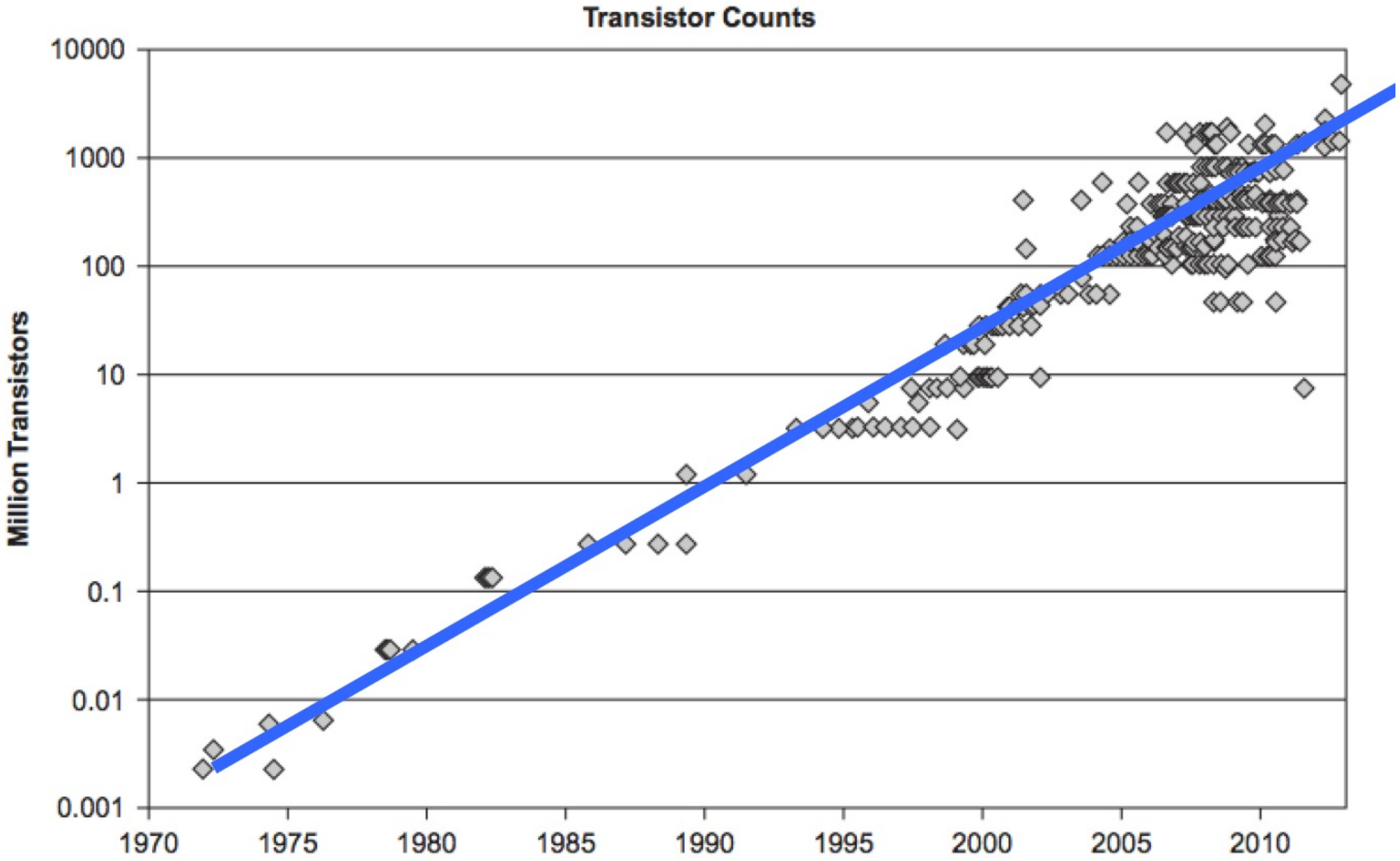


Moore's «Law»

- How many more transistors on a chip?
- Effect on speed of processor

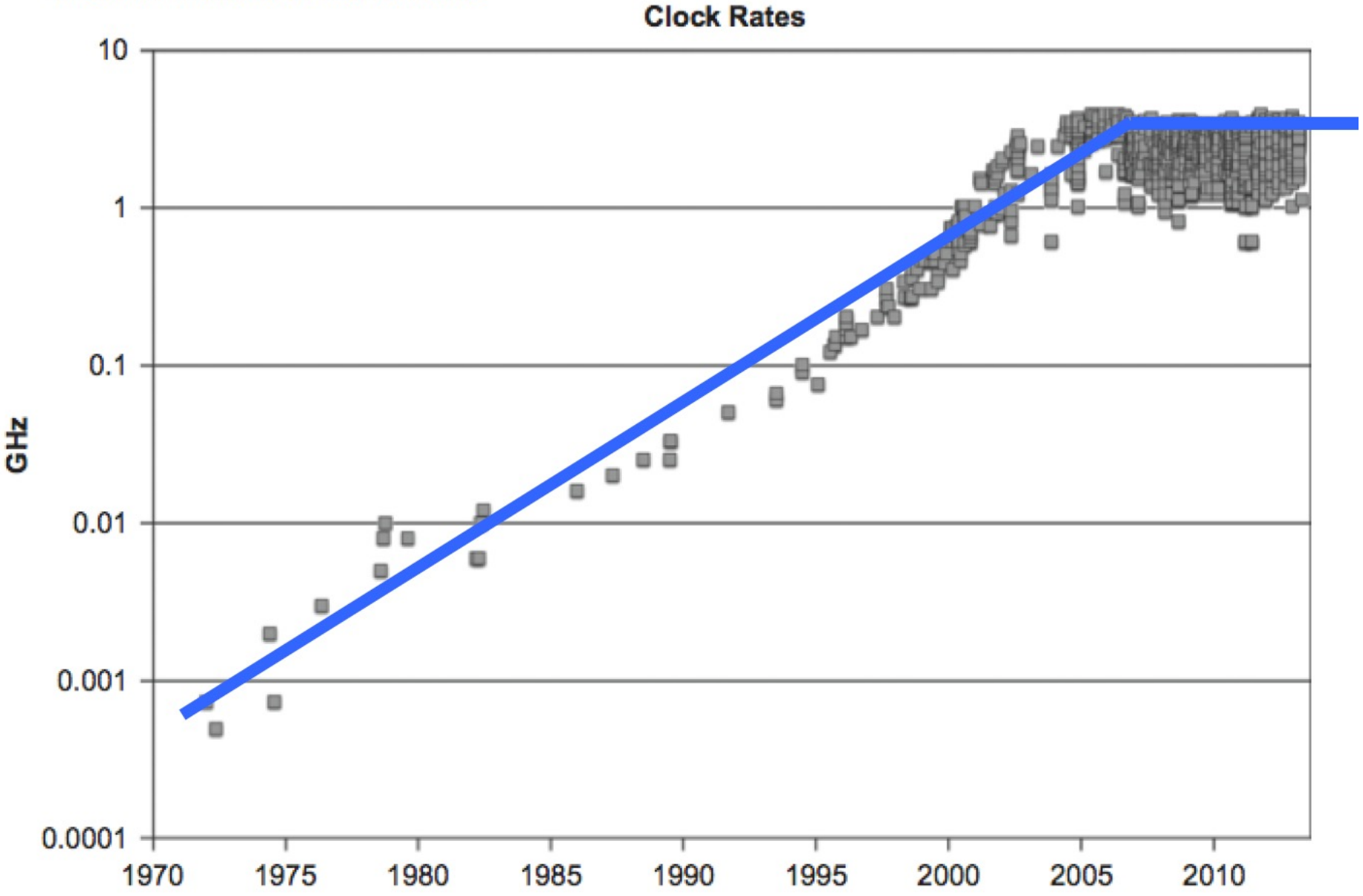
Transistors per Processor over Time

Continues to grow exponentially (Moore's Law)

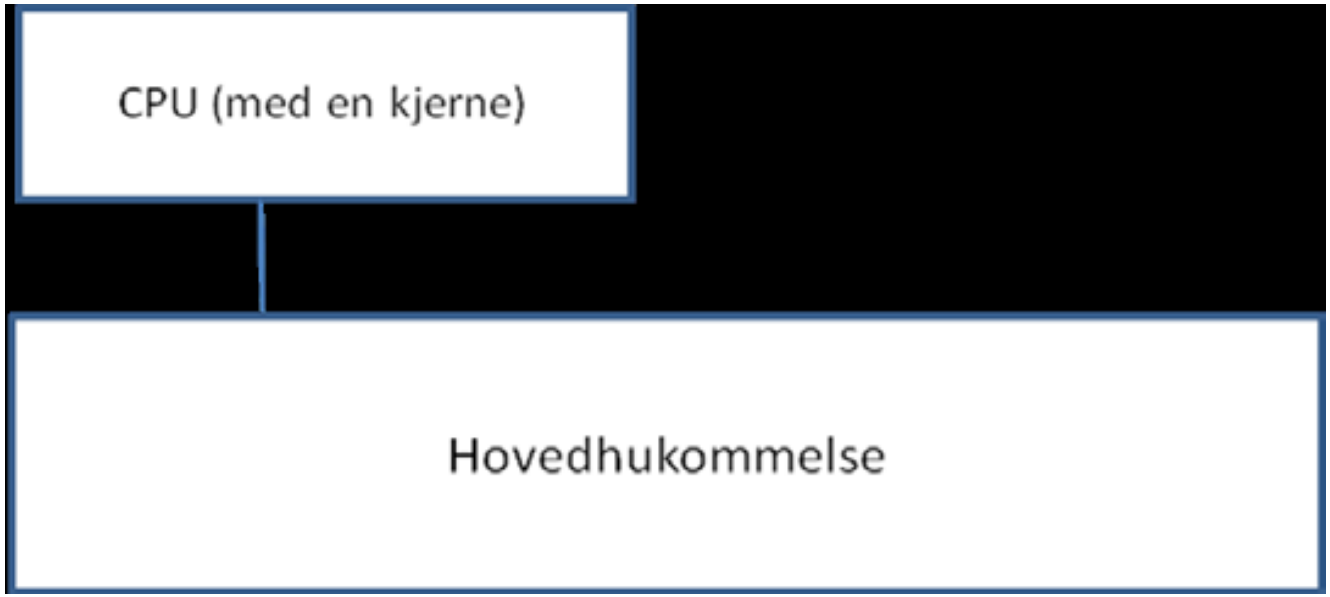


Processor Clock Rate over Time

Growth halted around 2005

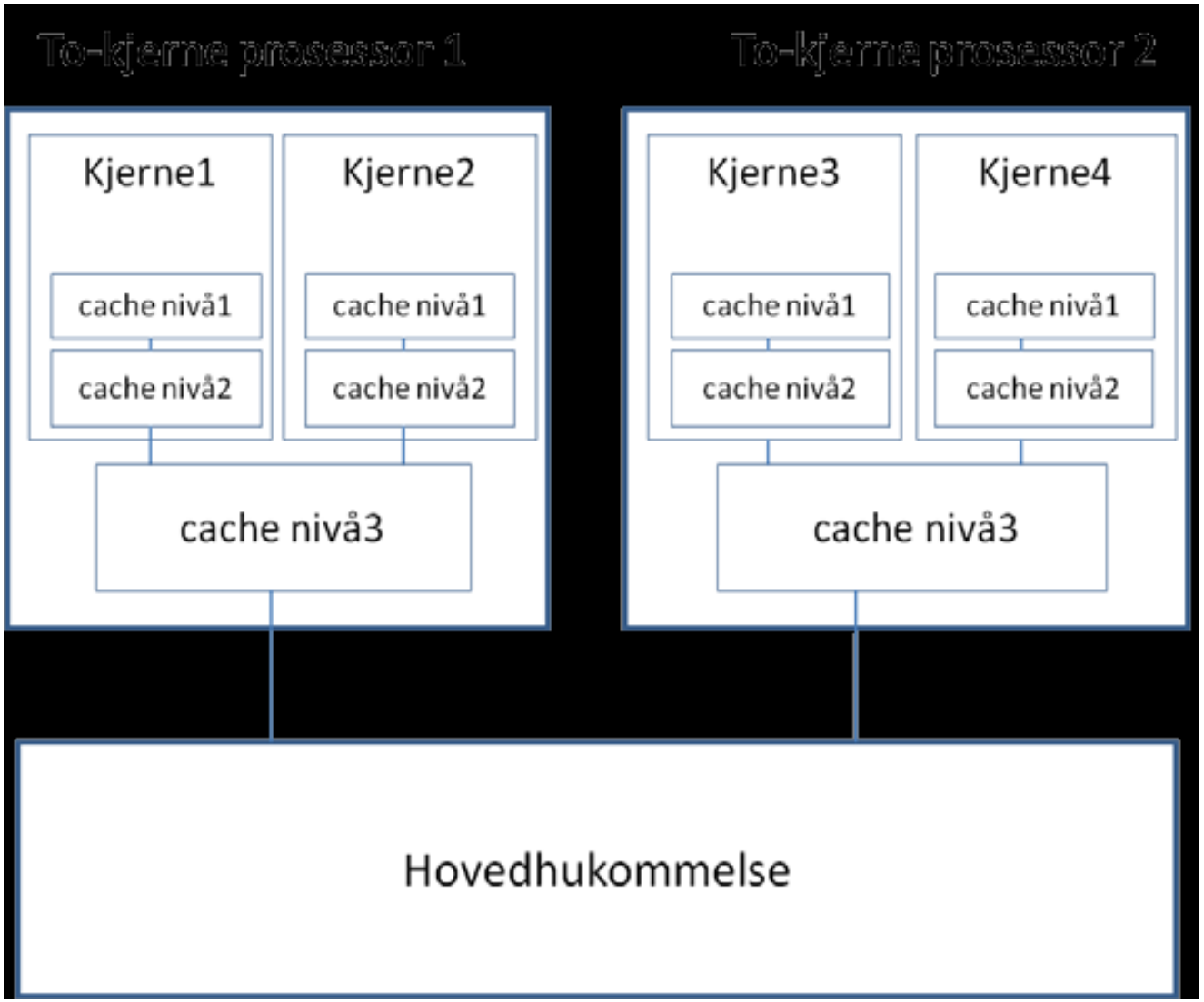


Maskin 1980 (uten cache)

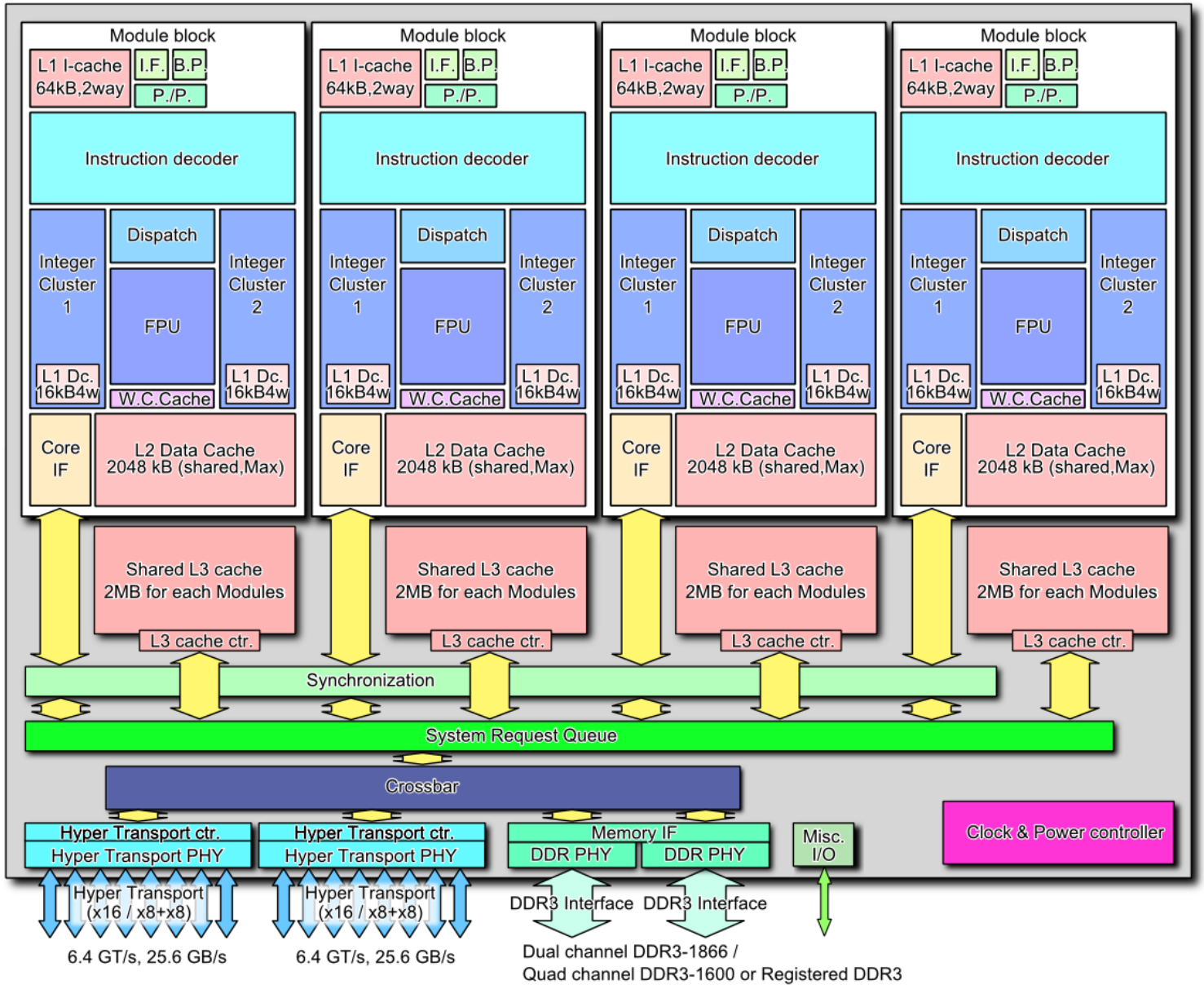


Figur 19.1 Skisse av en datamaskin i ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variable) direkte i hovedhukommelsen. Intel 8080: 1 MHz CPU

Maskin ca. 2010 med to dobbeltkjerne CPU-er



Hukommelses-systemet i en 4 kjerne CPU – mange lag og flere ulike beregningsmoduler i hver kjerne.:

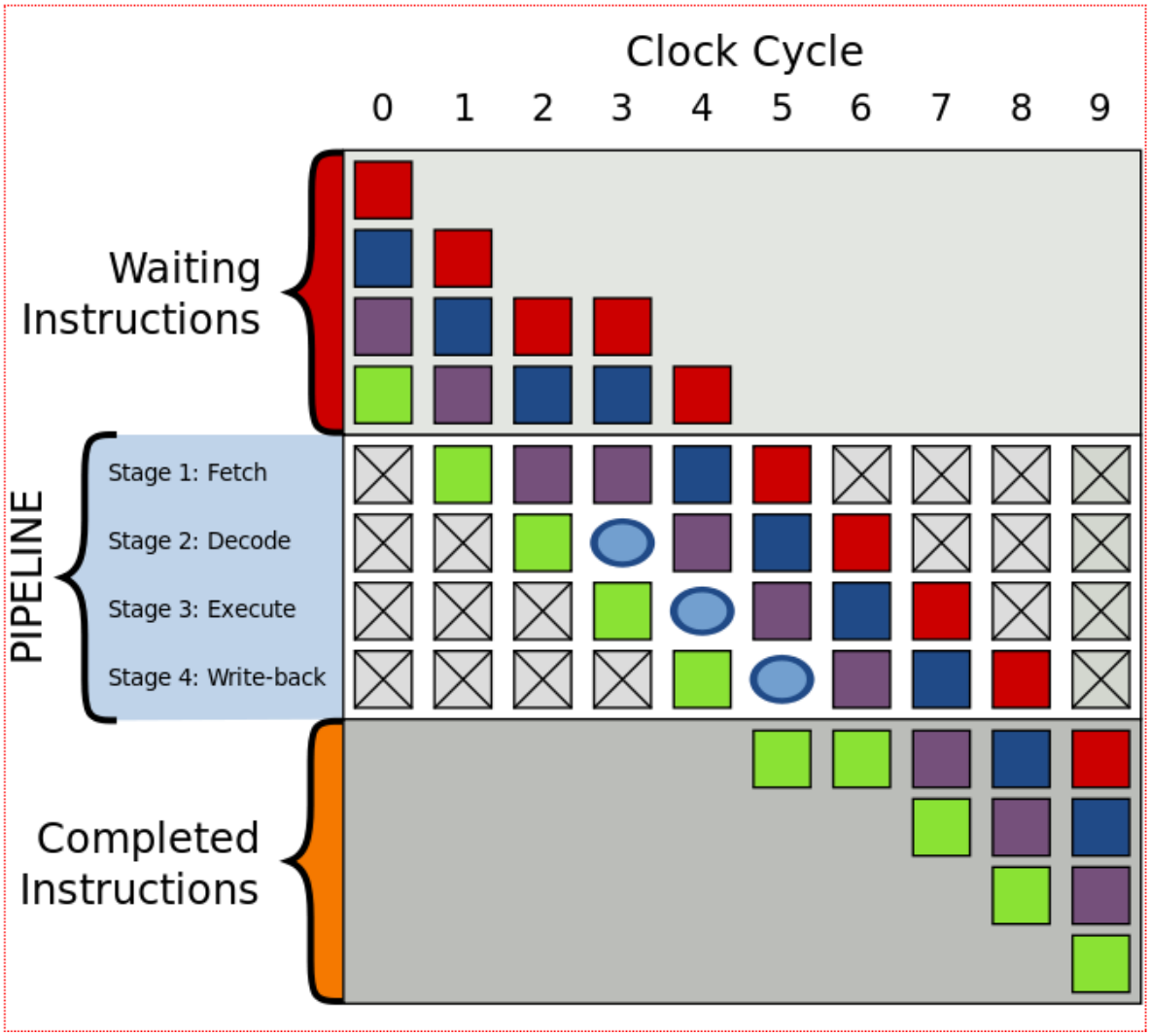


Hvordan tar vi hensyn til cache-systemet for å få raskere programmer?

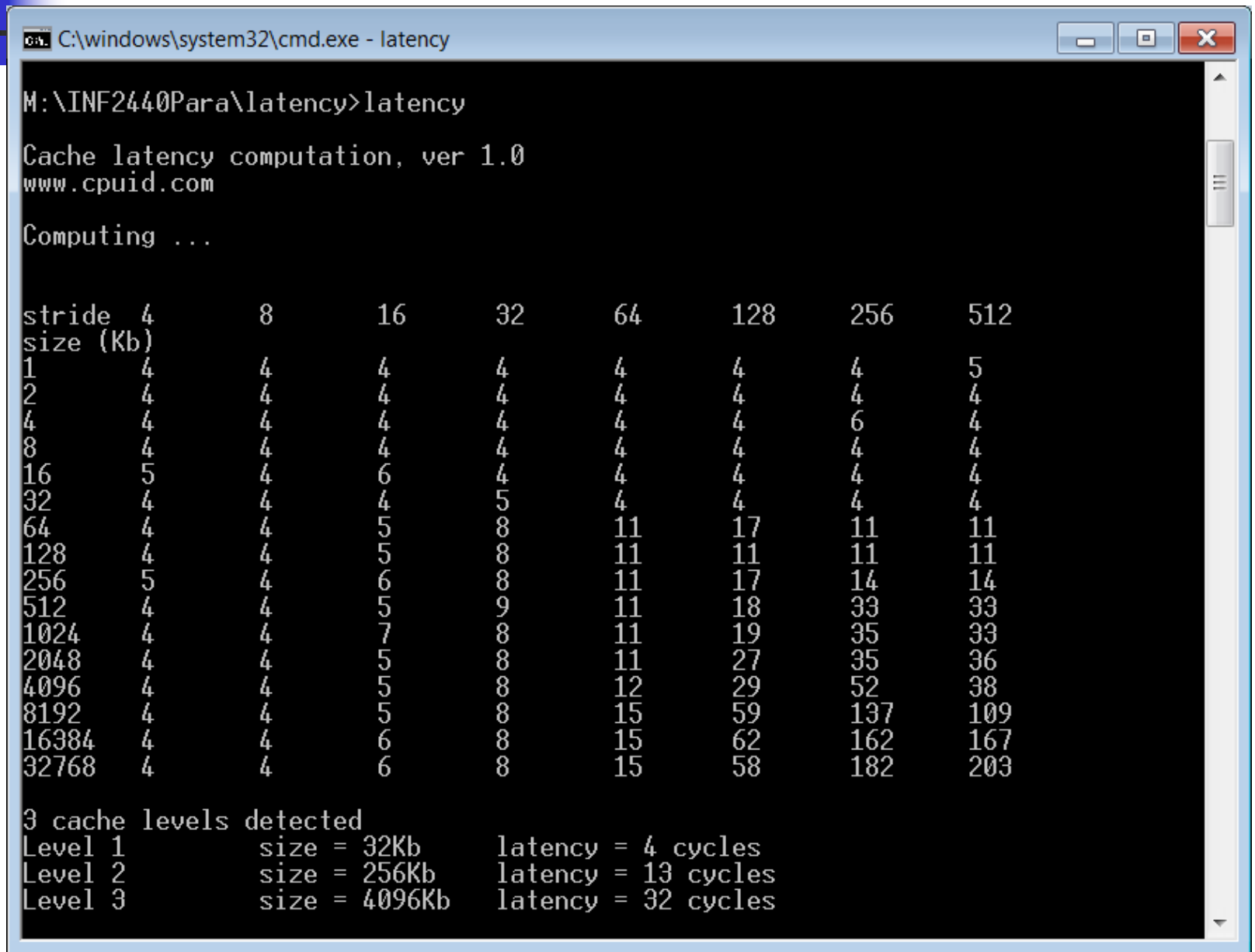
- Vi ser bare på data-cachene (lite å hente på instruksjonene)
- Viktig å vite er at hver gang vi skal hente data i hovedlageret , får vi en cach-linje = 64 byte = f.eks 8 heltall (int)
- Det er svært begrenset plass i cachene, og en cach-linje som ikke har vært brukt på 'lenge' vil bli 'kastet ut'(overskrevet av en annen, nyere) cache-linje
- Slik er raskest:
  - Jobber på få data (korte deler av en array) 'lenge' av gangen – ikke hoppe rundt.
  - Helst gå forlengs eller baklengs gjennom data (arrayene) (i, i+1,.. eller: i, i-1,..)

Vi må lage slike cache-vennlige programmer !

Instruksjonsparallellitet i en CPU-kjerne. Pipeline – flere instruksjoner (her 4) utføres *samtidig* i raskest mulig rekkefølge.



Test av forsinkelse i data-cachene og hovedhukommelsen - latency.exe (fra CPUZ)



Oppsummering – ideen om at vi har *uniform* aksesstid i hukommelsen er helt galt

- Hukommelses-systemet i en multicore CPU ,Intel Core i5-459 3.3 GHz, – mange lag (typisk aksesstid i instruksjonssykler):

1. Registre i kjernen (1) – 8/32 registre
2. L1 cache (3-4) – 32 Kb
3. L2 cache (13) – 256 kb
4. L3 cache (32) – 8Mb
5. Hovedhukommelsen (virtuell hukommelse) (ca. 200) – 8-64 GB
6. Disken (15 000 000 roterende) = 5 ms – 1000 GB – 1-5 TB
   FlashDisk (ca 2 000 000 les, ca. 10 000 000 skriv) = ca. 1 ms

Helt avgjørede for oss – cache-hukommelse

- Hva er cache
  - Raskere (men også dyrere) hukommelse mellom hovedlageret og kjernene.
  - Vi må ha cache fordi det er så store hastighetsforskjeller mellom en CPU-kjerne og hovedlageret ('main memory')
  - Ofte nå 3-4 lag med cache hukommelser + et antall registere i kjernen (enda raskere enn cache-hukommelsene) som holder data eller intruksjoner
  - Når en kjerne trenger data eller en ny instruksjon (og den ikke har det i et register) leter den nedover i cache-hukommelsene. Først cache level 1 (L1), så L2 cachen, .. , før den går til hovedhukommelsen for data eller instruksjoner.
  - Det finns flere teknikker for å gjøre dette raskt (som pre-fetch , dvs at systemet henter neste data/instruksjon uten at kjernen eksplisitt har bedt om det)

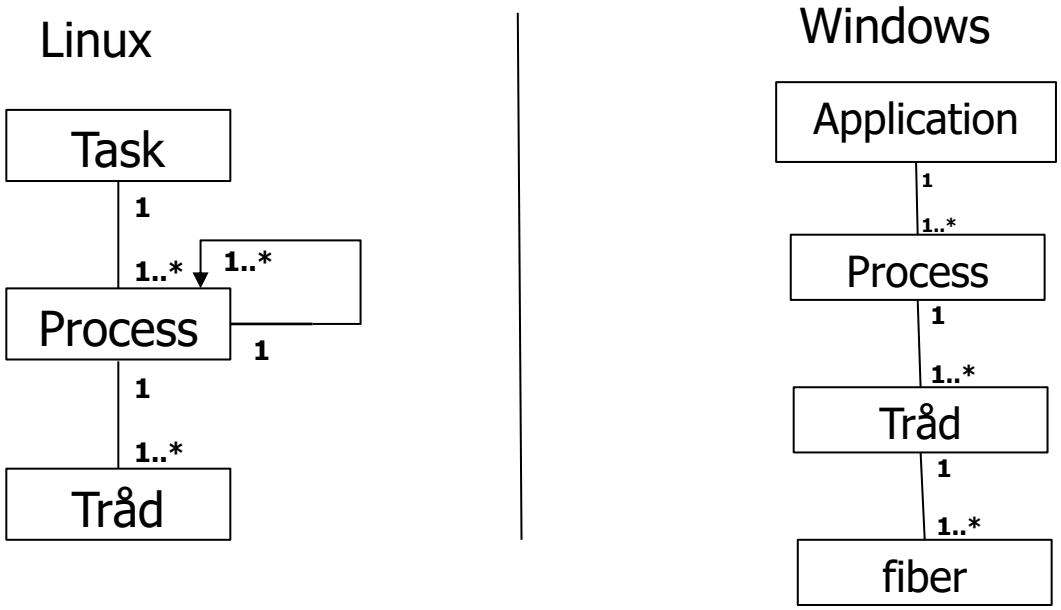
Vi kan ikke hente mer fra automatisk forbedring av hastigheten på våre programmer:

- **Ikke raskere maskiner** – luftkjølingsproblemet
- **Hovedhukommelsen - både** *mye* langsommere enn CPU-ene (derfor cache), og det å sette stadig flere kjerner oppå en langsom hukommelse gir køer.
- **Instruksjons-parallelliteten** i hver kjerne (pipelinen) er fult utnyttet – ikke mer å hente
- **Kompilatoren** – Java (etter ver 1.3) kompilerer videre til maskinkode og (etter ver 1.6) optimaliserer mye. JIT-kompilering. Ikke mulig å gjøre særlig mer effektiv

⇒ **Konklusjon** Skal vi ha raskere programmer, må vi som programmerere *selv* skrive parallelle løsninger på våre problemer.

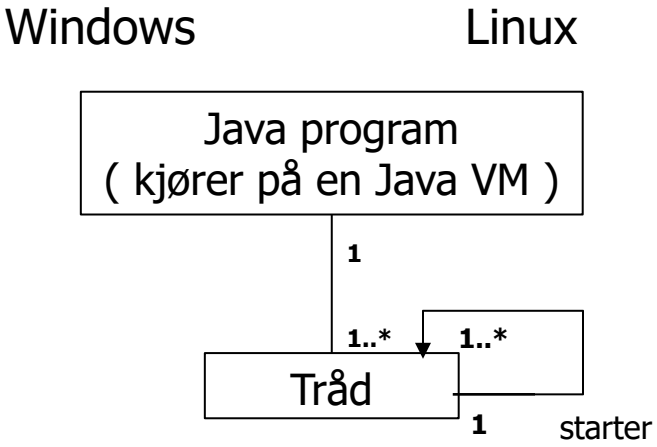
Operativsystemet og tråder

- De ulike operativsystemene (Linux, Windows) har ulike begreper for det som kjøres; mange nivåer (egentlig flere enn det som vises her)

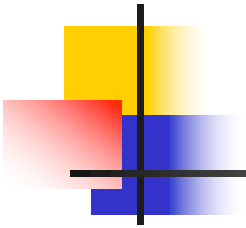


Heldigvis forenkler Java dette

Java forenkler dette ved å velge to nivåer



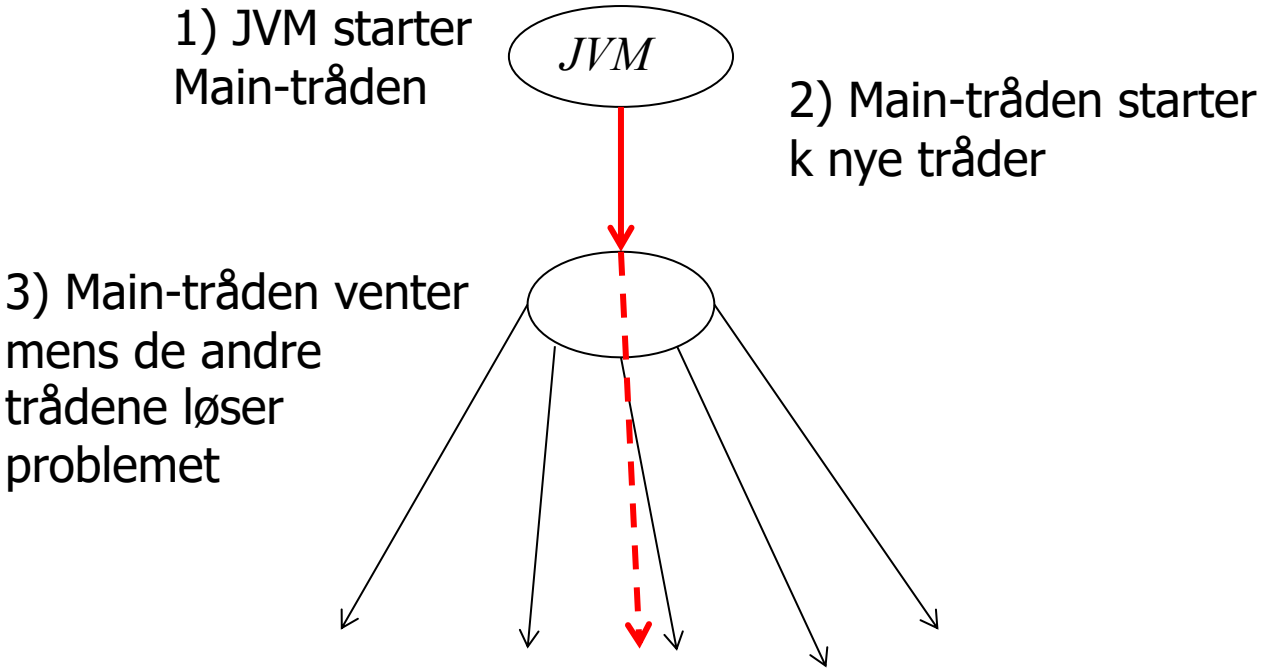
- **Alle trådene i et Java-program deler samme adresserom** (= samme plasser i hovedhukommelsen). Alle trådene kan lese og skrive i de variable (objektene) programmet har og ha adgang til samme kode (metodene i klassene).



Hva er tråder i Java ?

- I alle programmer kjører minst en tråd – main tråden (starter og kjører i public static void main).
- Main-tråden kan starte en eller flere andre, nye tråder.
- Enhver tråd som er startet, kan stoppes midlertidig eller permanent av:
  - Av seg selv ved kall på synkroniseringsobjekter hvor den må vente
  - Den er ferdig med sin kode (i metoden run), terminerer da
- Main-tråden og de nye trådene går i parallell ved at:
  - De kjører enten på hver sin kjerne
  - Hvis vi har flere tråder enn kjerner, vil klokka i maskinen sørge for at trådene av og til avbrytes og en annen tråd får kjøretid på kjernen.
- Vi bruker tråder til å parallellisere programmene våre

>java (også kalt JVM) starter main-tråden som igjen starter nye tråder



Tråder i Java er objekter av klassen Thread.

Konstruktør til Thread-klassen

Thread

`public Thread(Runnable target)`

Allocates a new `Thread` object. This constructor has the same effect as `Thread (null, target, gname)`, where `gname` is a newly generated name. Automatically generated names are of the form `"Thread-"`+*n*, where *n* is an integer.

Parameters:

`target` - the object whose `run` method is invoked when this thread is started. If `null`, this classes `run` method does nothing.

- **`Runnable target`** er :
  - En klasse som implementerer grensesnittet 'Runnable'
- Det er en annen måte å starte er tråd hvor vi lager en subklasse av Thread (ikke fullt så fleksibel).

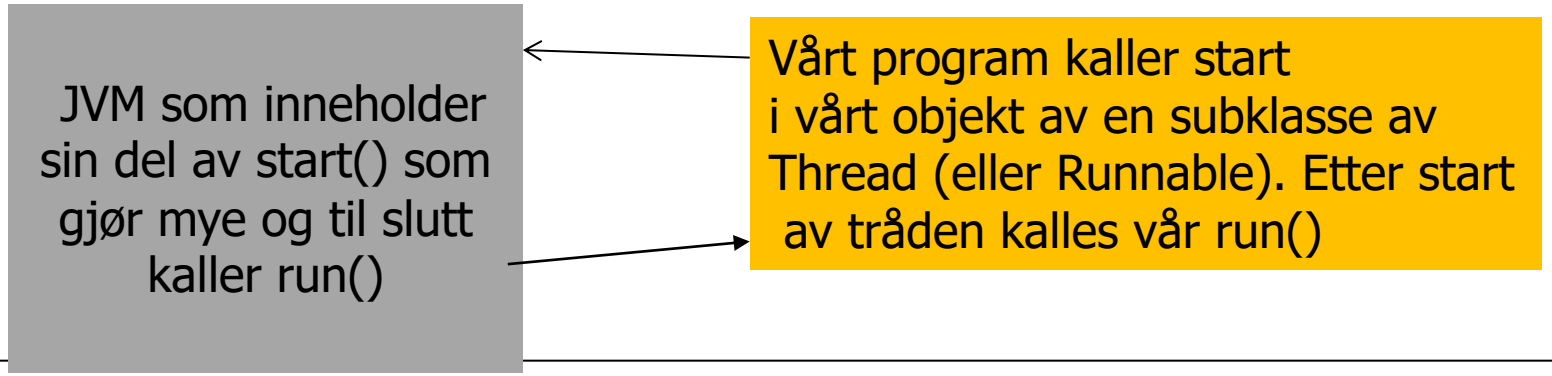
Tråder i Java

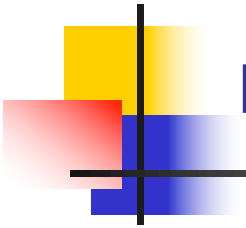
- Er én programflyt, dvs. en serie med instruksjoner som oppfører seg som ett vanlig, sekvensielt program – og kjører på én kjerne
- Det kan godt være (langt) flere tråder enn det er kjerner.
- En tråd er ofte implementert i form av en indre klasse i den klassen som løser problemet vårt (da får trådene greit aksess til **felles data**):

```
import java.util.concurrent.*;
class Problem { int [] fellesData ; // dette er felles, delte data for alle trådene
        public static void main(String [] args) {
                Problem p = new Problem();
                        p.utfoer();
                }
                void utfoer () { Thread  t = new Thread(new Arbeider());
                    t.start();
                 }

            class Arbeider implements Runnable {
                int i,lokalData; // dette er lokale data for hver tråd
                    public void run() {
                            // denne kalles når tråden er startet
                    }
             } // end indre klasse Arbeider
} // end class Problem
```

Tråder i Java

- En tråd er enten subklasse av Thread eller får til sin konstruktør et objekt av en klasse som implementerer Runnable.
- Poenget er at begge måtene inneholder en metode:
  - ’ public void run()’
- Vi kaller metoden start() i klassen Thread . Det sørger for at JVM starter tråden og at ’run()’ i vår klasse deretter kalles.





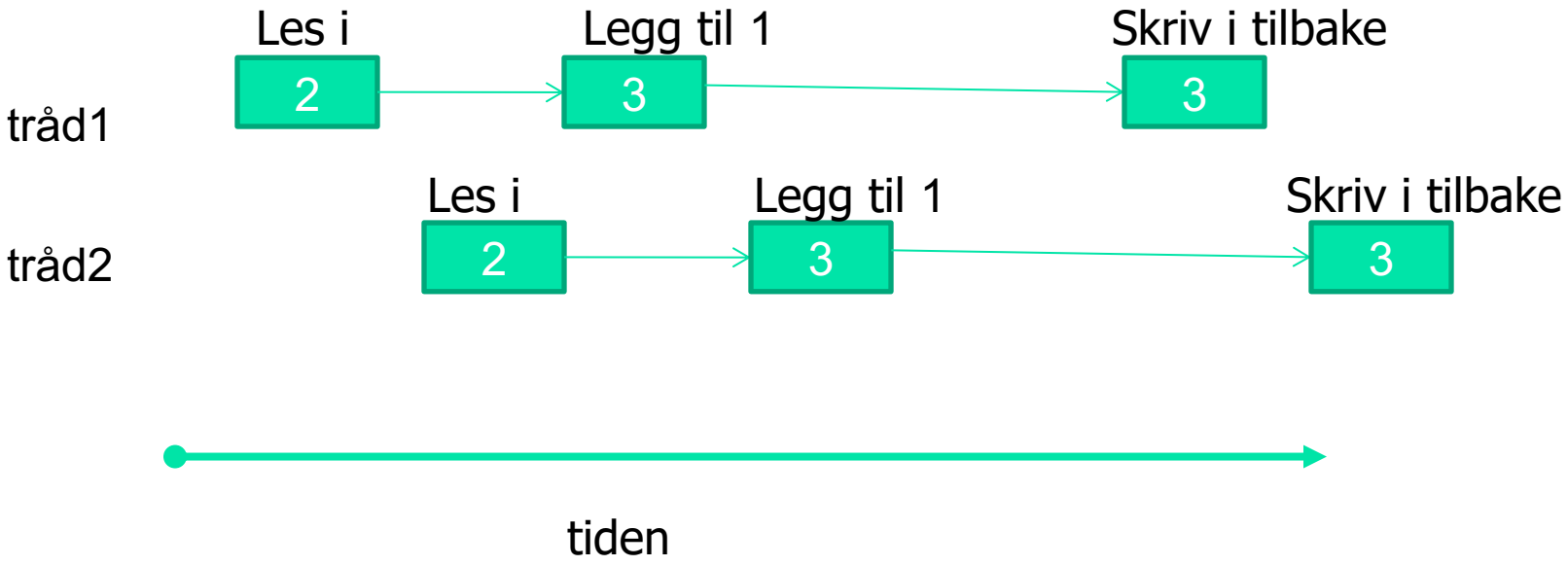
Flere problemer med parallellitet og tråder i Java

1. Operasjoner blandes (oppdateringer går tapt).
2. Oppdaterte verdier til felles data er ikke alltid synlig fra alle tråder (oppdateringer er ikke synlige når du trenger dem).
3. Synlighet har ofte med cache å gjøre.
4. The Java memory model (= hva skjer 'egentlig' når du kjører et Java-program).

- Vi må finne på 'skuddsikre' måter å programmere parallelle programmer
  - De er kanskje ikke helt tidsoptimale
  - Men de er lettere å bruke !!
  - Det er vanskelig nok likevel.
- Bare oversiktelige, 'enkle' måter å programmere parallelt er mulig i praksis

1) Ett problem i dag: operasjoner blandes ved samtidige oppdateringer

- Samtidig oppdatering - flere tråder sier gjentatte ganger: **i++ ;** der i er en felles int.
  - **i++** er 3 operasjoner: a) les i, b) legg til 1, c) skriv i tilbake
  - Anta i =2, og to tråder gjør i++
  - Vi kan få svaret 3 eller 4 (skulle fått 4!)
  - Dette skjer i praksis !



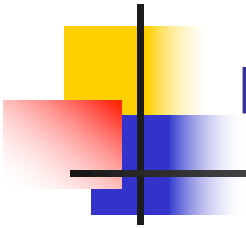
Test på i++; parallell

- Setter i gang **n tråder** (på en 2-kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger uten synkronisering;

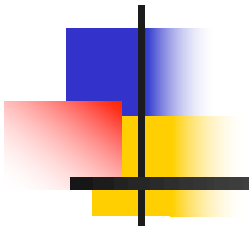
```
for (int j =0; j< 100000; j++) {
        i++;
}
```

- Vi fikk følgende feil - antall og %, (manglende verdier).
  Merk: Resultatene *varierer også mye* mellom hver kjøring :

Antall tråder n		**1**	**2**	**20**	**200**	**2000**
Svar	1.gang	100 000	200000	1290279	16940111	170127199
	2.gang	100 000	159234	1706068	16459210	164954894
Tap	1.gang	0 %	0%	35,5%	15,3%	14,9%
	2. gang	0%	20,4%	14,6%	17,7%	17,5%



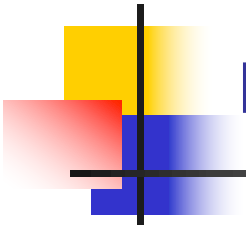
End of L1 IN3030 Lecture 2023-01-25



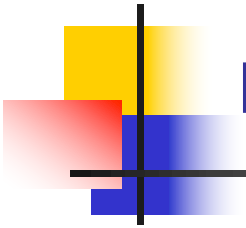
IN3030, L3, våren 2023 – Regler for parallelle programmer, mer om cache

Eric Jul
Programming Technology Group
Programming Section
Department of Informatics
University of Oslo

Hva har vi sett på i L02

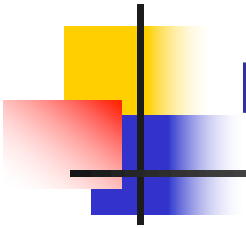


- Tråde i Java
- Én stygg feil vi kan gjøre: Samtidig oppdatering (skriving) på delte data (eks: to tråde skriver på samme variabel: `i++`)
- Synkronisering er vanskelig!
- Samtidig skriving på en variabel ***må synkroniseres***:
  - Alle objekter kan nyttes som en synkroniseringsvariabel, og da kan vi bruke enten en `synchronized` metode (treigt) for å gjøre det,
  - eller objekter av spesielle klasser som:
    - **CyclicBarrier**
    - **Semaphore**
    - **AtomicInteger**
    - En av MANGE andre mulige



Hva har vi sett på i L02 (fortsatt)

- I) Tre måter å avslutte tråder vi har startet.
  - join(), Semaphore og CyclicBarrier.
- II) Ulike synkroniseringsprimitiver
  - Vi skal bare lærte oss noen få - ett tilstrekkelig sett

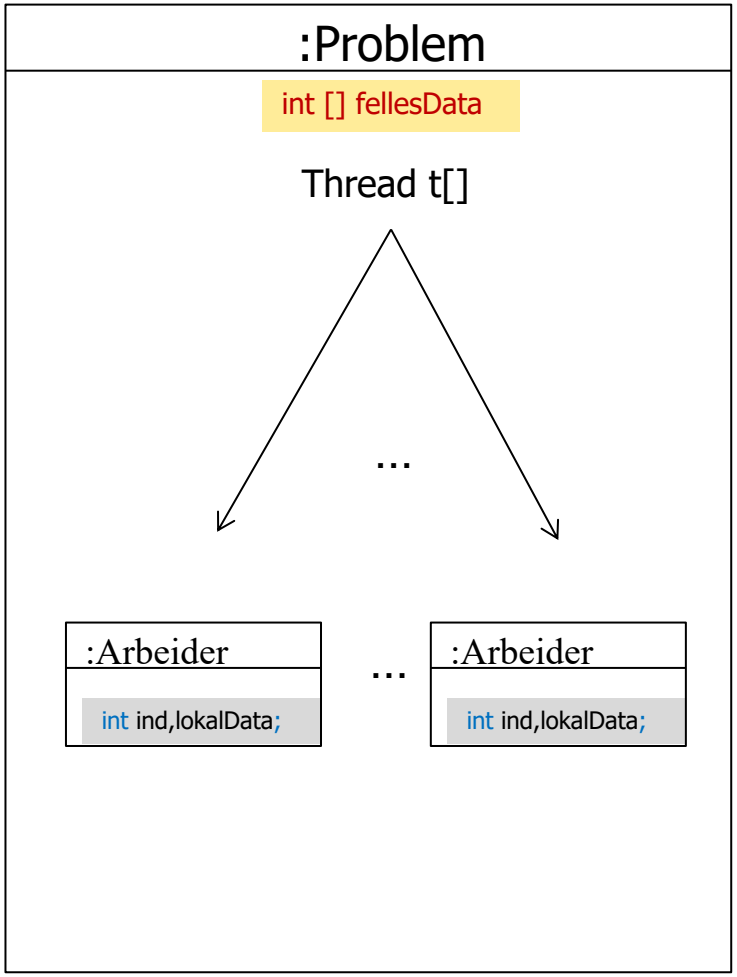


Plan for L03

- Modell(er) for hvordan vi programmerer
- Viktige regler om lesing og skriving på felles data.
- Synlighetsproblemet – «memory is not memory»
  - Hvilke verdier ser ulike tråder som leser variable som en annen tråd skriver på?
- Effekten på eksekveringstid av cache
- Matrixmultiplisering.

Modell for parallelle programmer

```
import java.util.concurrent.*;
class Problem { int [] fellesData , //  felles data
      public static void main(String [] args) {
            Problem p = new Problem();
            p.utfoer(12);
      }
      void utfoer (int antT) {
            … // utfør sekvensiell kode med tidtaking
          Thread [] t = new Thread [antT];

          for (int i =0; i< antT; i++)
              ( t[i] = new Thread(new Arbeider(i))).start();

          for (int i =0; i< antT; i++) t[i].join();
      }
      //.. metoder for sekvensielt  & parallell

      class Arbeider implements  Runnable {
        int ind, lokaleData; // lokale data
        //.. metoder for parallelt problem
         Arbeider (int in) {ind = in;)
            public void run(int ind) {
                  // kalles når tråden er startet
             } // end run
      } // end indre klasse Arbeider
} // end class Problem
```



Dette gjør at programmet blir mer effektivt

- I et virkelig brukerprogram vil vi ha testen:

```
if (n < LIMIT ){ løsXSekvensielt(param)
 } else {
```

  - I IN3030 skal vi ikke ha denne testen fordi vi er mer interessert i å se når en parallell løsning er langsommere **og** når den er raskere.
  - Vi kan si vi bestemmer LIMIT for ulike problemer:
    - For FinnMax er LIMIT ca = 1 mill.
    - For andre problemer er LIMIT langt lavere, f.eks senere vil vi se: 40 000 for sortering og 150 for matrise-multiplikasjon.
  - I sekvensielle programmer, som sortering gjøres også en slik test og man bruker 'innstikkSortering', hvis n < 32.
    - Arrays.sort() – som er Quicksort, bruker LIMIT = 47

Sekvensielt for små problemer: Slik skal virkelige programmer se ut (**ikke** i kurset)

```
class ProblemX{
    <felles data>

     <type> løsX(...) {
     if (n < LIMIT ){ løsXSekvensielt(param)
     } else {
        <start tråder. De løser hver sin del av
          problemet og tilsammen hele problemet>;
        <vent på at trådene er ferdige>;
        <hent svaret i felles data og returner>
    } // end løsX

    class ArbeiderTråd extends Thread{
       <Lokale data for en tråd>;
       ArbeiderTråd (param) {
          <lokale data = param>;
       public static void run() {
          <her løses denne trådens del av problemet
            i ett eller flere steg med synkronisering
             mellom hvert steg når vi bruker parallell kode>;
        } // end run
    } // end ArbeiderTråd
} // end class ProblemX
```

Konvensjoner som gjør at programmet ikke blir forkert - I

1. Alle arbeider-tråder har en lokal variabel: int indeks (=0,1,2,..,antTråder-1)
2. Vi antar at brukere som kaller løsX-metoden, kjører på main-tråden.
   - Dårlig idé er å la en tråd i et 'annet' parallelt problem kalle på en parallell løsning som løsX. Blir fort for mange tråder og treigt. (dvs. ikke parallelliser inne i en allerede parallellisert kode)
3. Vi lar trådene løse **hele** problemet.
4. Main-tråden bare initierer felles data og starter hver tråd - før den legger seg og venter på at trådene blir ferdige.
   Da er hele problemet løst og ligger i felles data.
5. Problemet som arbeider-trådene skal løse, kan bestå av ett eller flere steg. Vi synkroniserer da alle arbeider-trådene med en CyclicBarrier mellom hvert av stegene.

(fortsettes neste foil)

Konvensjoner som gjør at programmet ikke blir forkert - II

6. Må ett av stegene (f.eks det siste) være sekvensielt, lar vi bare tråd med indeks == 0 gjøre det:

```
if (indeks == 0) {
   < Gjør det sekvensielle steget før neste synkronisering>;
}
```

   De andre arbeider-trådene går her bare rett til neste barrier-synkronisering (eller avslutning).
7. Hvis behovet for å ha en enkel seksvensiell kode oppstår midt under beregningene, kan alle trådene regne ut samme svar uten synkronisering seg imellom (skjer f.eks i parallell Quicksort)
8. Arbeider-trådene initierer bare lokale variable i sin konstruktør.
   8. Husk at objektet ikke er ferdig når konstruktøren kjører. Mye galt kan skje (se boka JCiP kap 3.2) hvis andre tråder får en peker til objektet før det er ferdig.
   9. Ingen kall til andre metoder i konstruktøren.
   10. Kan forebygges: Lad ALLE tråde passer en cyclisk barrier etter initialisering.
9. All handling i arbeider-trådene skjer i run() og i metoder kalt fra run().

VIGTIKT Konvensjon: Tre avgjørende prinsipper for lesing og skriving på felles data.

- Før (og etter) synkronisering på felles synkroniserings-objekt gjelder :
  A. Hvis ingen tråder skriver på en felles variabel, kan alle tråder lese denne.
  B. To tråder må aldri skrive samtidig på en felles variabel (eks. `i++` går galt)
  C. Hvis bare én tråd skriver på en variabel må også bare denne tråden lese denne variabelen før synkronisering – ingen andre tråder må lese den før synkronisering.

Muligens ikke helt tidsoptimalt, men enkel å følge – gjør det mulig å skrive parallelle programmer.

Har vist pkt. A og B, skal nå vise pkt. C

Synlighetsproblemet (hvilke verdier ser ulike tråder som leser variable som en annen tråd skriver på)

- Lage et testprogram som har:
  - To **felles** variable. int a,b;
  - To klasser, arbeider-tråder SkrivA og SkrivB,
  - en som øker a & en øker b (100 000 ganger) og skriver ned verdiene av a og b i hver sin *lokale* arrayer: mA[] og mB[] (antså to sett av disse).

```
for (int j = 0; j<antGanger; j++) {
        a++;
        mA[j] =a;
        mB[j] =b;
 }
```

  - og en annen tråd som tilsvarende øker b

```
for (int j = 0; j<antGanger; j++) {
        b++;
        mA[j] =a;
        mB[j] =b;
 }
```

Ytre klasse SamLes med to indre klasser SkrivA og SkrivB

```
public class SamLes{
      int  a=0, b=0;                  // Felles variable a , b
      CyclicBarrier sync, vent ; // begge starter 'samtidig'
      int  antGanger ;
      SkrivA aObj;
      SkrivB bObj;


void utskrift() {  ...   };

void  utfor () {

              vent = new CyclicBarrier((int)antTraader+1);
               sync = new CyclicBarrier((int)antTraader);

              (aObj =  new SkrivA()).start();
              (bObj =  new SkrivB()).start();

               try{
                  // main venter på aObj og bObj ferdige
                     vent.await();
              } catch (Exception e) {return;}
        utskrift();
 } // utfor
```

```
class SkrivA extends Thread{
      int [] mB = new int[antGanger],
             mA = new int[antGanger];
      public void run() {
                try {  // wait on the other thread
                              sync.await();
                } catch (Exception e) {return;}

                for (int j = 0; j<antGanger; j++) {
                      a++;
                       mA[j] =a;
                       mB[j] =b;
                 }
                 try {  // wait on the other thread + main
                               vent.await();
                 } catch (Exception e) {return;}
     } // end run A
} // end class Para

class SkrivB extends Thread{
      int [] mB = new int[antGanger],
             mA = new int[antGanger];
      public void run() {
                try {  // wait on the other thread
                              sync.await();
                } catch (Exception e) {return;}

                for (int j = 0; j<antGanger; j++) {
                      b++;
                       mA[j] =a;
                       mB[j] =b;
                 }
                 try {  // wait on the other thread + main
                               vent.await();
                 } catch (Exception e) {return;}
     } // end run B
} // end class SamLes
```

Hva tester vi her ?

- Ser på om de to trådene (aObj og bObj) alltid ser oppdaterte verdier av den andre variabelen (ser f.eks objA at b er helt oppdatert) ?
- Utskrift vanskelig: Selv om starter nesten likt, må de synkroniseres på utskrift (og ikke skrive ut alt!):

Leter utover i de to arrayene: mA[] i de to objektene aObj og bObj og starter utskrift ut når a-verdiene i er like og > 0, og skriver da ut de 10 neste verdiene av a og b i aObj og bObj

Resultater: Er det feil her (gamle verdier, e.l)

= like verdier i a og b

SkrivA			SkrivB		
a.mA[722]= 723	a.mB[722]= 1458	|	b.mA[1457]= 723	b.mB[1457]= 1458
a.mA[723]= 724	a.mB[723]= 1458	|	b.mA[1458]= 724	b.mB[1458]= 1459
a.mA[724]= 725	a.mB[724]= 1460	|	b.mA[1459]= 725	b.mB[1459]= 1460
a.mA[725]= 726	a.mB[725]= 1460	|	b.mA[1460]= 726	b.mB[1460]= 1461
a.mA[726]= 727	a.mB[726]= 1461	|	b.mA[1461]= 727	b.mB[1461]= 1462
a.mA[727]= 728	a.mB[727]= 1463	|	b.mA[1462]= 728	b.mB[1462]= 1463

- NB. SkrivA (=aObj) har a-ene riktige (oppdatert) og SkrivB har b-ene oppdatert
- For eksempel. første og andre linje tvilsomme sammen:
  - A har akkurat økt a fra 722 til 723, og ser b som 1458, MEN
  - B har akkurat økt b fra 1458 til 1459, og ser a som 723
  - I neste linje ser A fortsatt b som 1458, men a i aObj er lik 724
- Dette kan bare forklares ved at A og B operasjonene blandes
- Vi vet ikke når b for aObj har en verdi (eks 1458 eller1460) hvilken a-verdi som hører til disse.
- Og noen verdier for b (1459, 1462) sees aldri av aObj, men bObj ser dem.
- **Konklusjon:** Ulike tråder kan se ulike verdier for felles variable og man vet ikke når en tråd har oppdatert (skrevet) på 'sine' variable og dette er synlig i annen tråd.

WTF: What Terrible Fiasco!

= like verdier i a og b

SkrivA			SkrivB		
a.mA[722]= 723	a.mB[722]= 1458	|	b.mA[1457]= 723	b.mB[1457]= 1458
a.mA[723]= 724	a.mB[723]= 1458	|	b.mA[1458]= 724	b.mB[1458]= 1459
a.mA[724]= 725	a.mB[724]= 1460	|	b.mA[1459]= 725	b.mB[1459]= 1460
a.mA[725]= 726	a.mB[725]= 1460	|	b.mA[1460]= 726	b.mB[1460]= 1461
a.mA[726]= 727	a.mB[726]= 1461	|	b.mA[1461]= 727	b.mB[1461]= 1462
a.mA[727]= 728	a.mB[727]= 1463	|	b.mA[1462]= 728	b.mB[1462]= 1463

DISASTER: Think CAREFULLY about what happened here!

MAYDAY, MAYDAY, MAYDAY:

*You MAY think that a core can »write to memory» BUT in reality there is no guarentee that others can see the new value right away!! It CAN take time – and in the meanwhile two cores can **READ DIFFERENT VALUES** from the **SAME** variabel!!!*

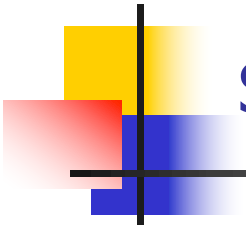
FIX: the two thread must synchronize first! ☺ Problem solved!

4) Skriving på **nærliggende** elementer i en array, kode.

```
class ParaArray{
                    int  []tall;
                    CyclicBarrier b ;
                    int antTraader, antGanger ;
....
class Para implements Runnable{
    int indeks;
   Para(int i) { indeks =i;}
       public void run() {
                    for (int j = 0; j< antGanger; j++) {
                          oekTall(indeks);
                     }
                    try {  // wait on all other threads + main
                         b.await();
                    } catch (Exception e) {return;}
       } // end run
        void oekTall(int i) { tall[i]++; }
  }
} // end ParaArray
```

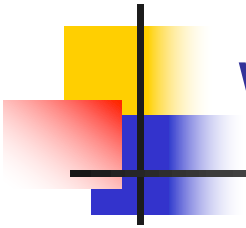
- Cache-linja er nå 64 byte (og en int er 4 byte)
- Går det greit med at flere tråder (indeks=0,1,…,k-1) skriver på a[tråd.indeks] mange ganger i parallell?
- Tester: Vi lageret program som gjør det :

```
>java ParaArray 10 100000000
Maskinen har 8 prosessorkjerner.
Tid 100000000 kall * 10 Traader =
0.032600 sek,
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
 sum:100000000, tap:0 =   0.0%
```



SAVED BY THE CAVALRY

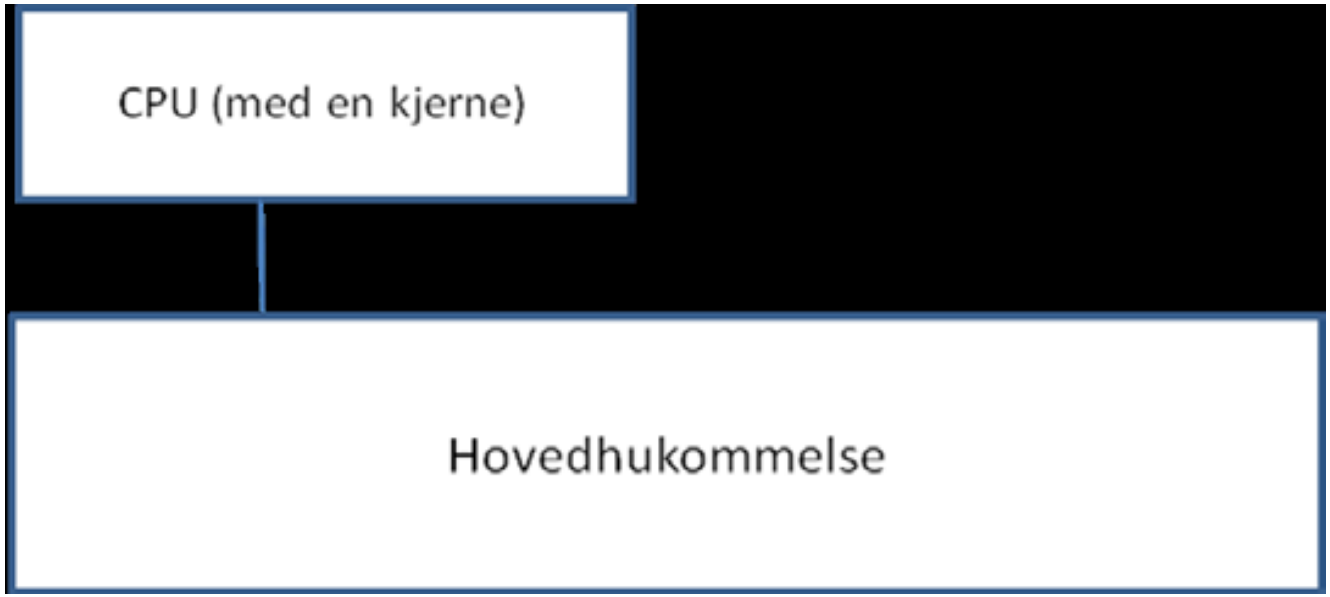
SAVED!: We can write to separate elements in an array WITHOUT synchronization!!



WHAT is a cache?

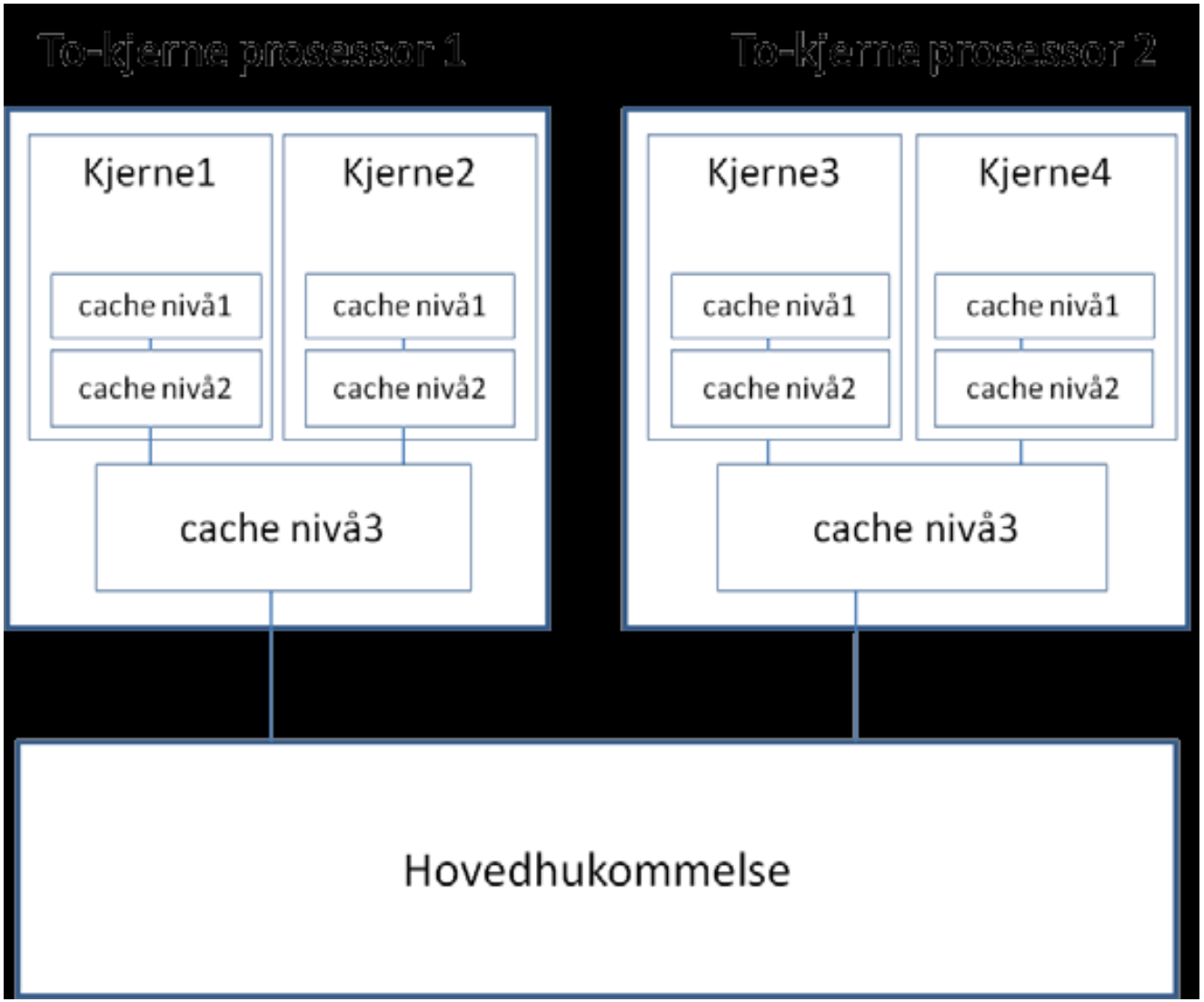
- Principle of a cache
- Speed of memory vs size
- Second reason: Speed of electricity:
  - Electricity travels about 6-7 cm per machine cycle on a 3 GHz CPU

Maskin 1980 (uten cache)

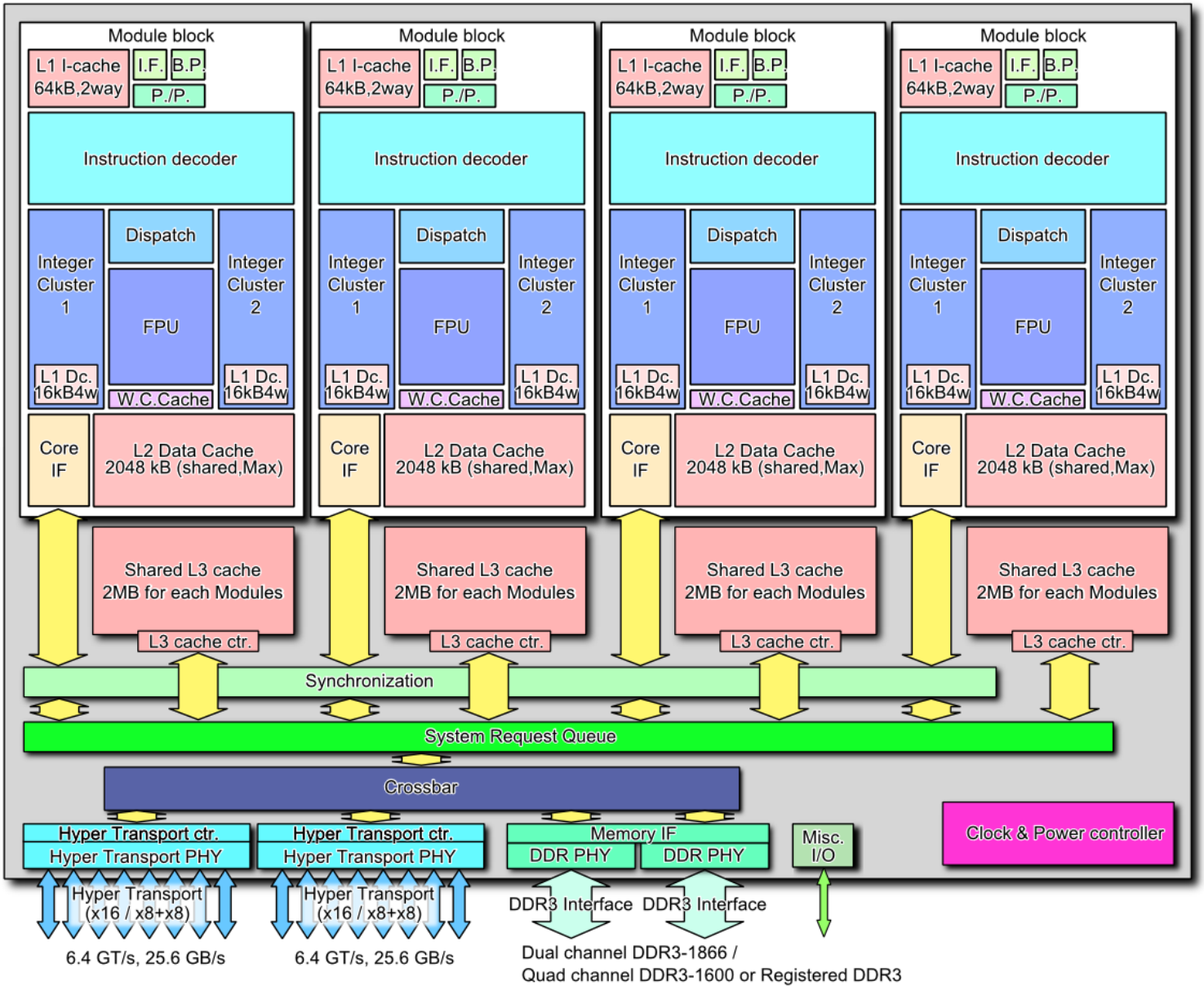


Figur 19.1 Skisse av en datamaskin i ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variable) direkte i hovedhukommelsen. Intel 8080: 1 MHz CPU

Maskin ca. 2010 med to dobbeltkjerne CPU-er



Hukommelses-systemet i en 4 kjerne CPU – mange lag og flere ulike beregningsmoduler i hver kjerne.:

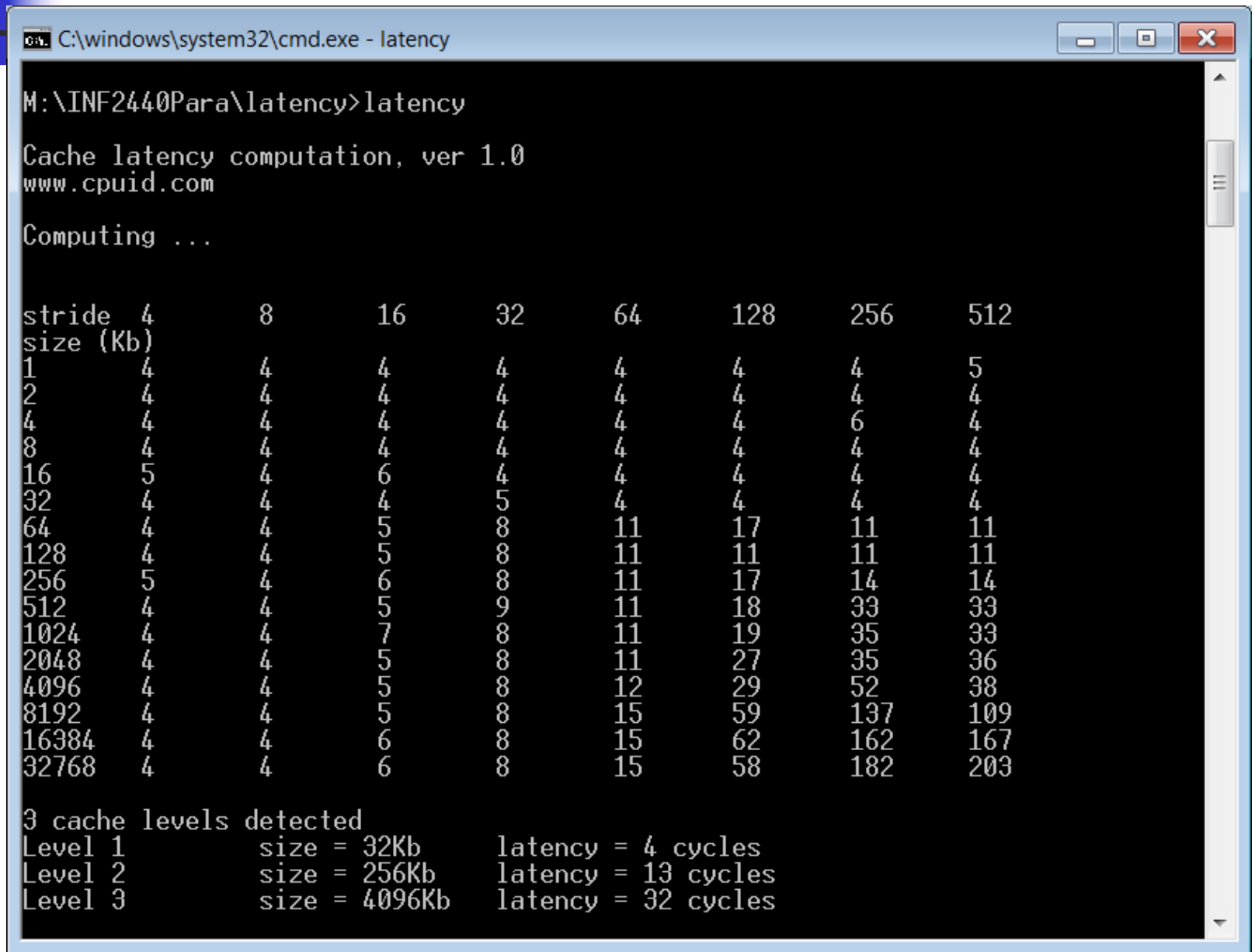


Hvordan tar vi hensyn til cache-systemet for å få raskere programmer?

- Vi ser bare på data-cachene (lite å hente på instruksjonene)
- Viktig å vite er at hver gang vi skal hente data i hovedlageret , får vi en cach-linje = 64 byte = f.eks 8 heltall (int)
- Det er svært begrenset plass i cachene, og en cach-linje som ikke har vært brukt på 'lenge' vil bli 'kastet ut'(overskrevet av en annen, nyere) cache-linje
- Slik er raskest:
  - Jobber på få data (korte deler av en array) 'lenge' av gangen – ikke hoppe rundt.
  - Helst gå forlengs eller baklengs gjennom data (arrayene) (i, i+1,.. eller: i, i-1,..)

Vi må lage slike cache-vennlige programmer !

Test av forsinkelse i data-cachene og hovedhukommelsen - latency.exe (fra CPUZ)



Oppsummering – ideen om at vi har *uniform* aksesstid i hukommelsen er helt galt

- Hukommelses-systemet i en multicore CPU ,Intel Core i5-459 3.3 GHz, – mange lag (typisk aksesstid i instruksjonssykler):

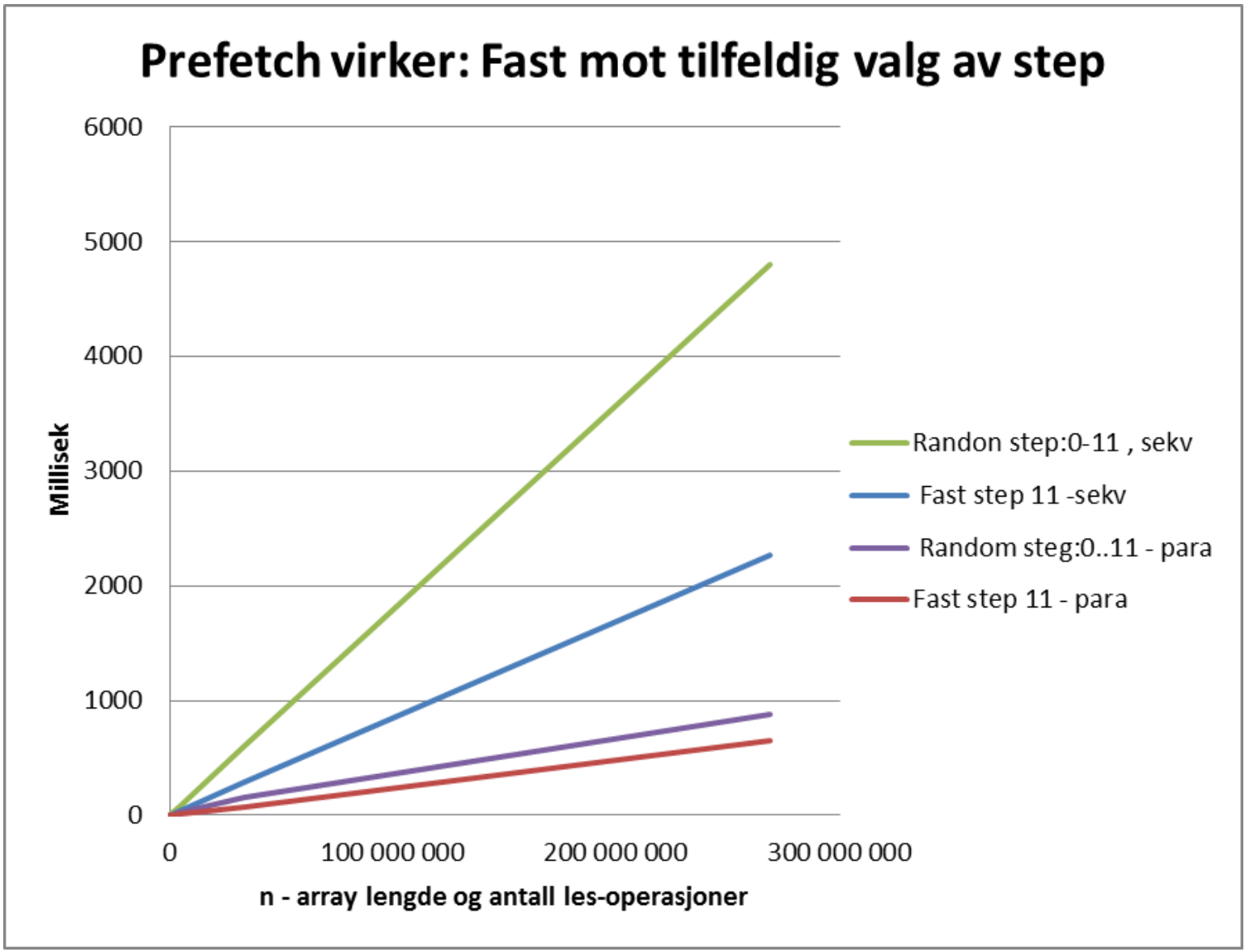
1. Registre i kjernen (1) – 8/32 registre
2. L1 cache (3-4) – 32 Kb
3. L2 cache (13) – 256 kb
4. L3 cache (32) – 8Mb
5. Hovedhukommelsen (virtuell hukommelse) (ca. 200) – 8-64 GB
6. Disken (15 000 000 roterende) = 5 ms – 1000 GB – 1-5 TB
   FlashDisk (ca 2 000 000 les, ca. 10 000 000 skriv) = ca. 1 ms

Prefetch-mekanismen på brikken

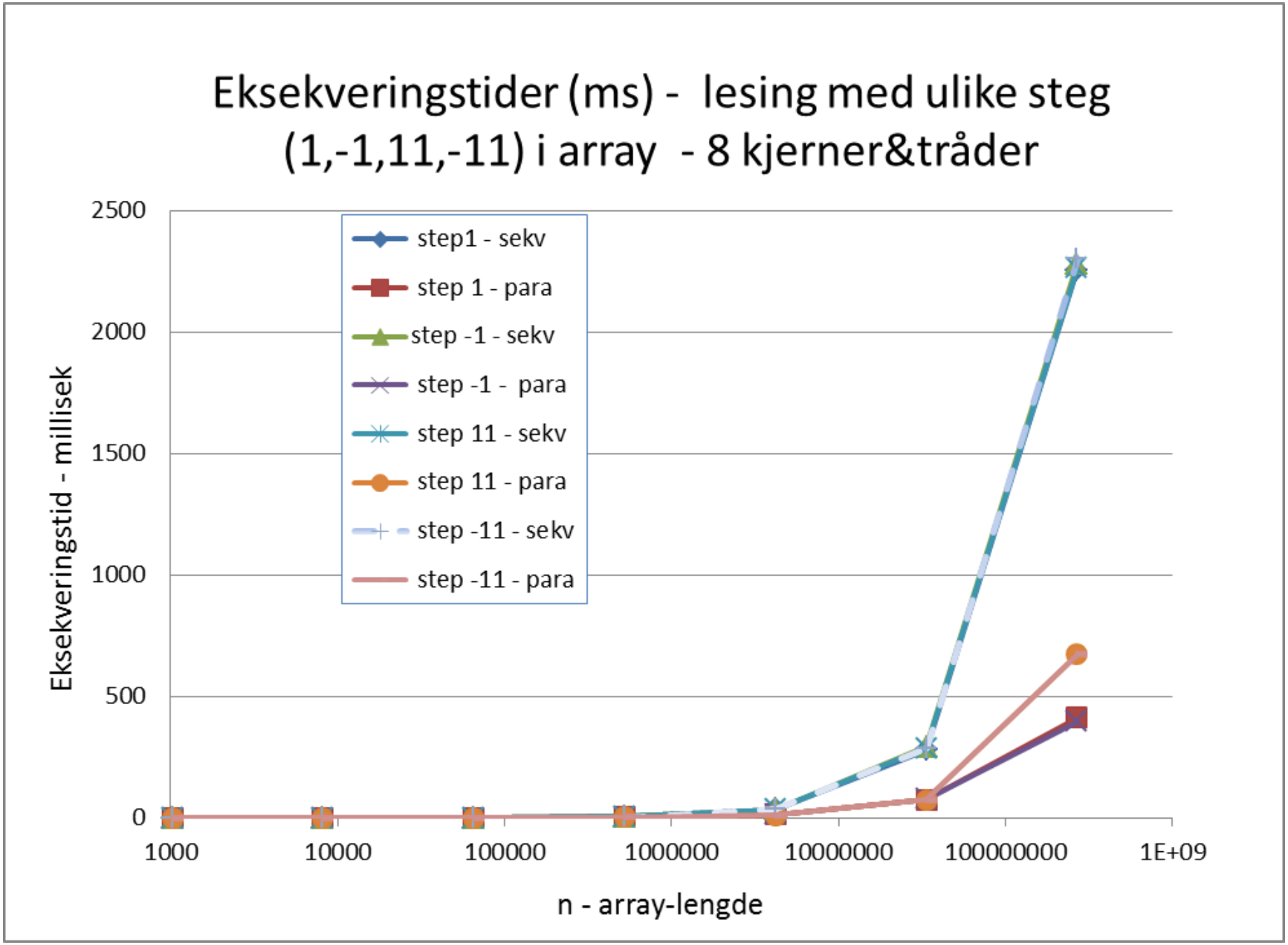
- Består i at hvis vi aksesserer (leser eller skriver) element i k i en array a[] og så element k+m, så prøver elektronikken å hente element a[k+2m], a[k+3m],…, ***før*** vi har bedt om det.
- Tillegget m kan være både positivt og negativt .
- Hvis elementene a[k+2m] ligger i samme cachelinje, går dette spesielt fort.
- Skrev testprogram for dette og testet m = 1, -1,11 og -11

```
for (int i=0;i < a.length; i++){
    // index = Math.abs((index+r.nextInt(step+1))%a.length);
    index = Math.abs((i+step)%a.length );
    sum += a[index];
}
```

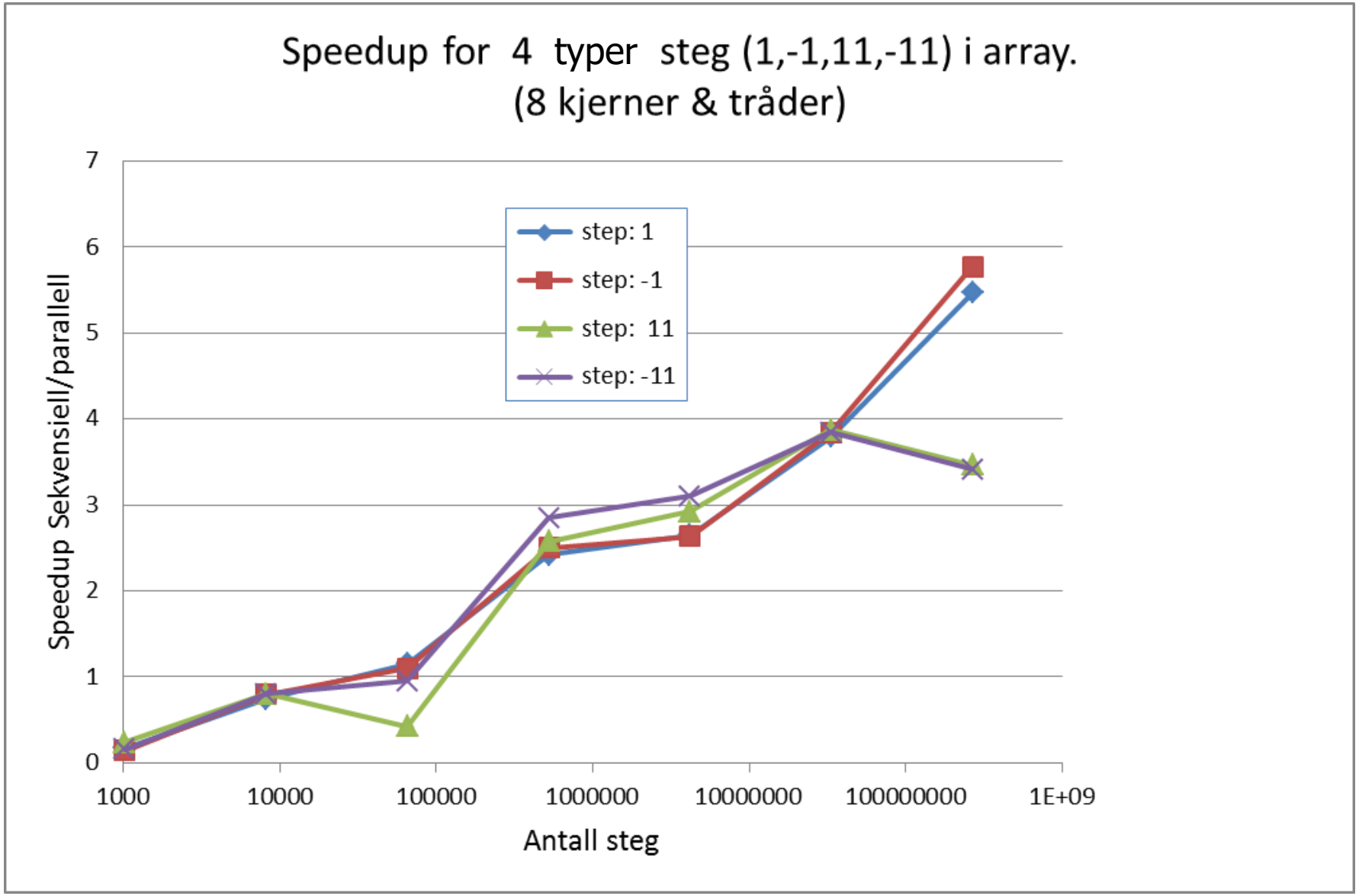
- Hva fant vi og hva kan vi slutte av det.
- Først grafer



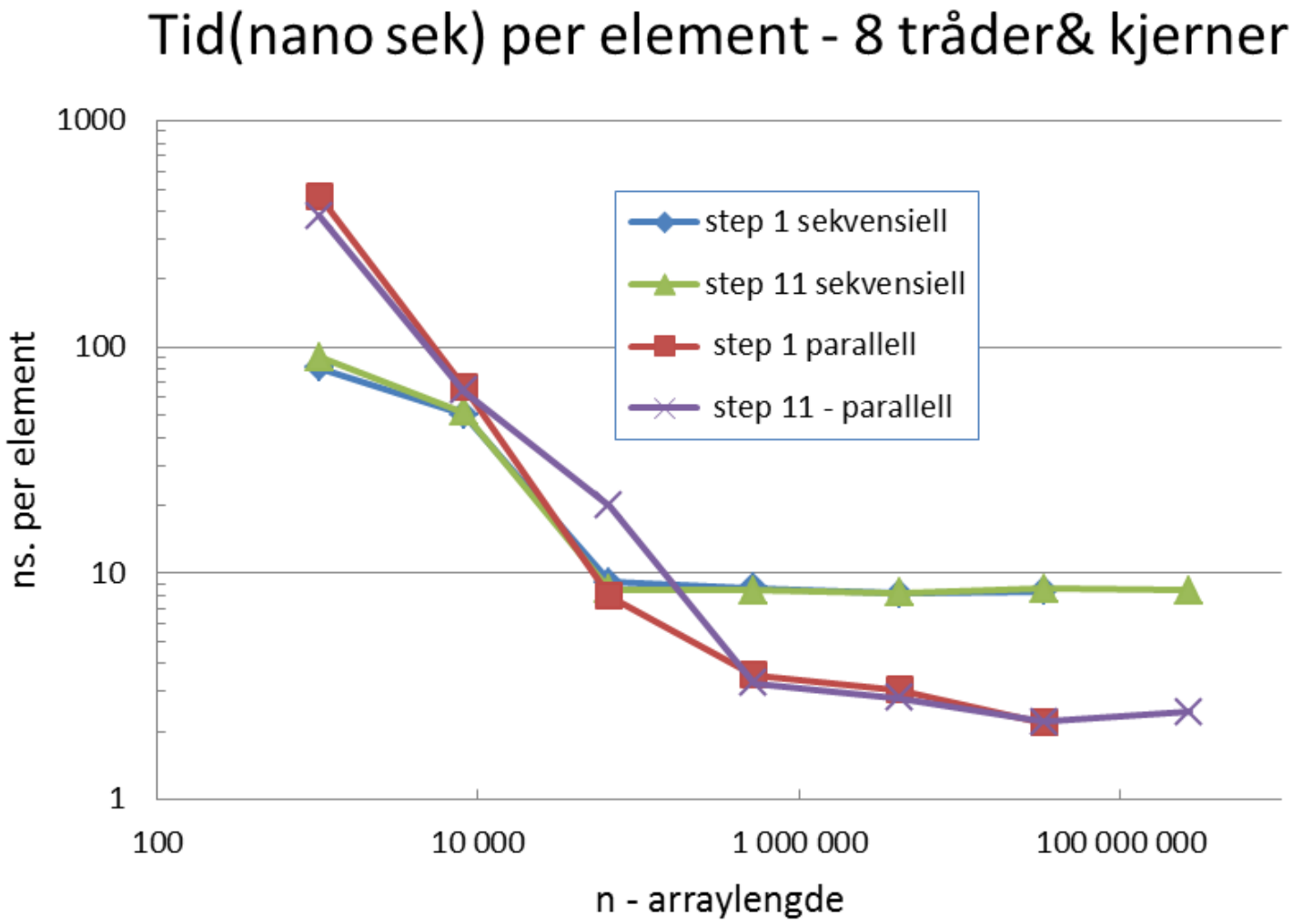
Konklusjon1: Fast steglengde er ca. dobbelt så raskt som tilfeldig (pga prefetch)



Konklusjon2: negative og positive tillegg er like raske.

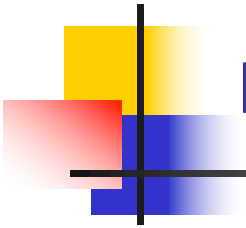


Konklusjon3: Spesielt steg 1,-1 har bra speedup pga. samme cachelinje



Konklusjon 4: Prosesseringstiden per element synker med økende n (JIT?)

Konklusjon 5: Per element tar det litt over 2*2,8 = ca. 6 instruksjoner å summere et element til en sum i parallell.



Prefetch-mekanismen hjelper en del

- Ikke så viktig som cache-systemet
- Ikke så viktig som JIT-kompilering
- men hjelper til og går på ingen måte i veien for de to viktigste mekanismene – ca. 2x raskere
- Programmet som laget data til disse grafene er laget av programmet Prefetch.java som er lagt ut på hjemmesida
- Grafene er laget i Excel (velg graftype:scatter diagram):
  - sett inn et slikt i regnearket og trykk så Select Data

Effekten på eksekveringstider av cache

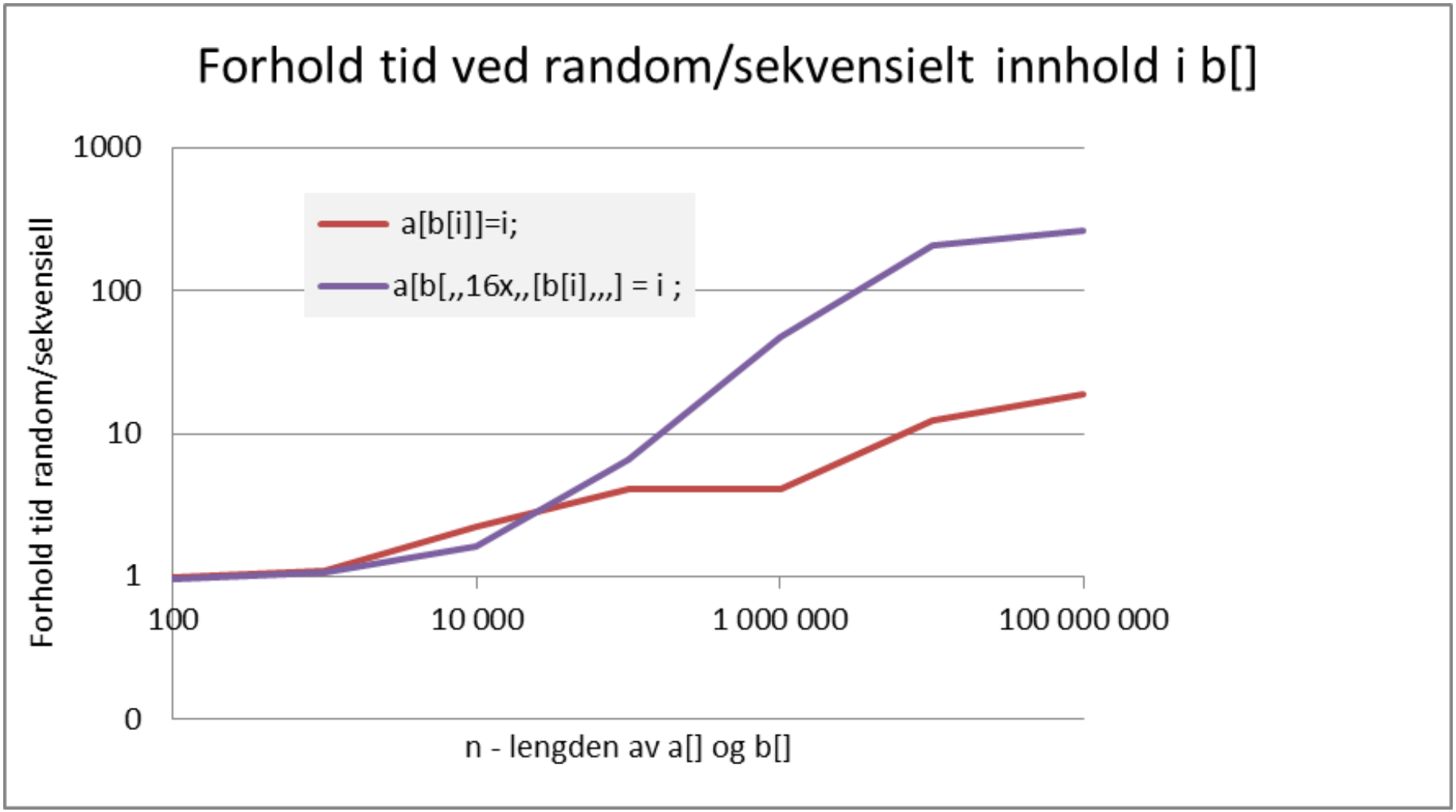
1) Hvor lang tid tar det å utføre n ganger (n=100, 1000, 10 000,….,100 mill):

   a[b[i]]=i;

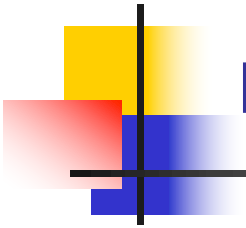
2) Avhenger av hva b[] inneholder:
   1) Hvis b[i] = i (sekvensiell) , så er a[b[i]] = a[i] og vi har 'alt' i cachen
   2) Hvis innholdet i b[] er tilfeldig trukket mellom 0:n-1, så er hver les/skriv i lageret en hopping frem og tilbake i a[] – ingen nytte av cachen

3) Neste graf viser hvor mange ganger lenger tid det tar å utføre ganger de to måtene å fylle b[]
   – enten b[i] = i, eller b[i] = random(0..n-1)

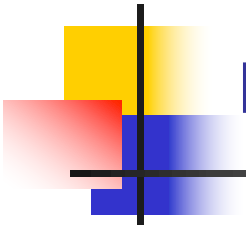
Hvor mange ganger tregere går random innhold i b[] enn b[] = 0,1,2,3,.. ?



Konklusjon – nestet aksess a[b[i]]

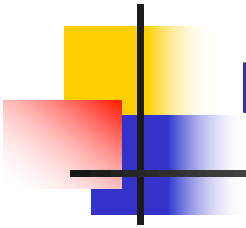


- For 'små' verdier av n <1000, gir cachen god aksess til både hele a[] (viktigst), og til b[].
- For store verdier av n > 100 0000 blir det meget langsommere, og vi kan få mellom 12 – 240 ganger langsommere kode (pga. cache-miss) når innholdet av b[] er 'tilfeldig'.
- Slike uttrykk a[b[i]] og a[b[c[i]]] finner vi i Radix-sortering som vi skal granske i en senere forelesning.



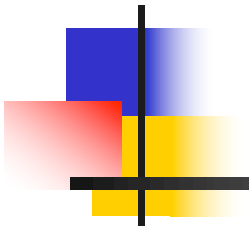
Matrix Multiplication

- (Blackboard)



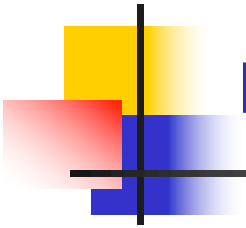
End L03v23

- End of lecture L03v23



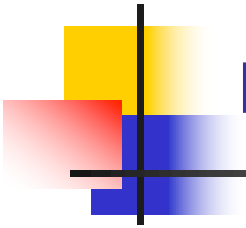
IN3030 L05v23 – About Moore's Law, the speed of light, JIT, and Gustavson

Eric Jul
Programming Technology Group
Department of Informatics
University of Oslo



Review F4

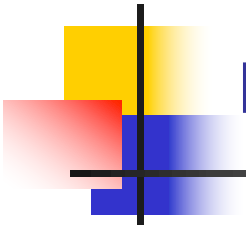
1. Oblig 2 Matrix multiply
2. Guest lecture on CPU's and GPU's



Plan for F5

- Synchronization is HARD and ERROR PRONE!

I. Moore's Law
II. Speed of light
III. Why distribution
IV. JIT compilation
v. How to present your timing results
VI. Hva er PRAM modellen - og hvorfor er den ubrukelig for oss
VII. Amdahl's and Gustavson's laws
VIII. PRAM Model



Moore's Law

- Used to be known as: «CPU speeds double every 2 years»

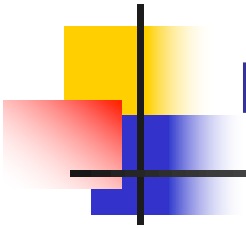
Original Moore's Law

- Transistors per square cm doubles:
- 1965 Article: Every year
- 1975 Article: Every two years

Perspektiv: Utvikling i CPU, minne, nettverk og mere om Moore's law

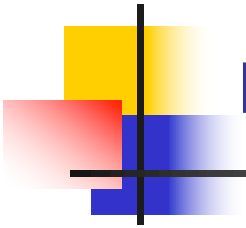
- 1980 CPU: Intel 8080 8-bit processor
  - «int»: one byte
  - «long»: two bytes
  - CPU clock frequency: 1 MHz
  - Memory: 64 kilobytes max, *i.e.*, 16 bit addresses (2 bytes)
  - 128 kB floppy disk
  - Data transmission 1.2 kbit/s == 150 bytes/s

- A look at Moore's law
  - Moore's original prediction 1965
  - Moore's revisted prediction 1975



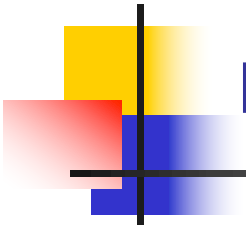
Moore's Law Summary of Effects

- 1958-1975
  - Doubling of transistors every year
  - Derived effect: doubling of CPU speeds
- 1975-2005
  - Doubling of transistors every two years
  - Derived effect: doubling of CPU speeds – screaching halt at about 3 GHz
- 2005-2020
  - Doubling of transistors every two years
  - Derived effect: doubling of number of CORES *or* doubling of the amount of on-chip cache



Newer Machines

- 2020: My Macbook Pro w Intel i7
  - 2.4 up-to 3.2 GHz – 8x cores
  - 16 Gbyte Memory
  - 10 Gbit/s network
  - 1 GB Solid State Disk (SSD)



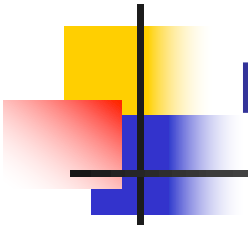
Newer Machines

- 2023: My Macbook Pro w Intel i7
  - 2.4 up-to 3.2 GHz – 12x cores: 1
  - 16 Gbyte Memory
  - 10 Gbit/s network
  - 1 GB Solid State Disk (SSD)

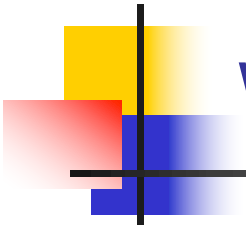
Performance 1980 vs 2018

- 2018: Intel i7
  - 1 MHz CPU vs 8 x 3.2 GHz ~ factor 25,600
  - 16 Gbyte Memory vs. 64 kbytes ~ factor 250,000
  - 10 Gbit/s network vs 1 kbit/s ~ factor 10,000,000
  - 128 kB floppy disk vs 1 TB SSD ~ factor 8,000,000
  - 1200 baud serial line vs 1 Gbit/s Ethernet ~ factor 800,000

- ☺

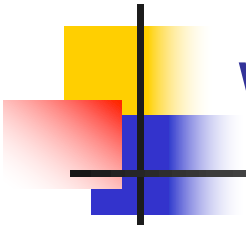


BREAK L05v23



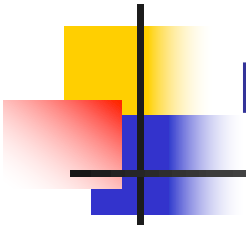
What is ping?

- `Ping`: low-level internet messaging: sends an empty message from one computer to another. The other computer returns it
- sending an empty message from Copenhagen/Oslo to Seattle & back
- In 1988, when the internet was first »opened» in Scandinavia: a ping took a little under 200 ms
- How long does this take 35 years later – in 2023?



What about ping time?

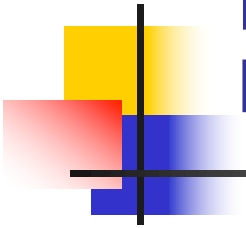
- Ping: sending an empty message from Oslo to Seattle & back
- How long does this take 1988 vs 2022?



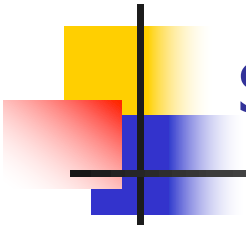
Round-trip ping Time

- 1988: Approximately 180-200 ms
- 2022: How much faster?
  - 1,000,000x faster?
  - 100,000x faster?
  - 10,000x faster?
  - 1,000x faster?
  - 100x faster?
  - 10x faster?
  - Same time?

Latency has ***not*** changed: dominated and limited by the *incredibly* slow speed of light!

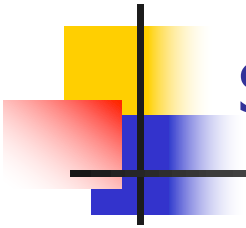


- How fast is the speed of light approximately?
- Great circle route over and back 16,000 km
- Speed of light in fiber 11/15, copper 3/5
- Great Circle round trip time: minimum of about 100 ms
- So ping time will ***NEVER*** go below 100 ms!



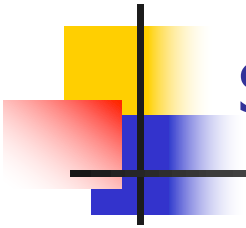
Speed of Light

- How fast is the speed of light approximately?
  - 300,000 km/s
  - Exact value?
- Speed of light
  - in fiber 11/15: about 220,000 km/s
  - in copper 3/5: about 180,000 km/s



Speed of light revisited

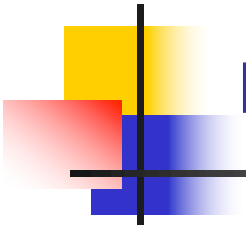
- How fast is the speed of light approximately?
  - 300,000 km/s
- How far does light travel in 1 ns?
- How far in copper in 1 ns?
- When a 3 GHz core executes one cycle, how far does light travel?



Speed of light: answers to questions

- How fast is the speed of light?
  - About 300,000 km/s
  - 299,792,458 m/s EXACTLY – by definition
- How far does light travel in 1 ns?
  - About 30 cm – call it a *lightfoot*
- How far in copper in 1 ns?
  - About 18 cm
- When a 3 GHz core executes one cycle, how far does light travel in vacuum and in copper?
  - About 10 cm (vacuum) and 6 cm (copper)

PRAM modellen for parallelle beregninger

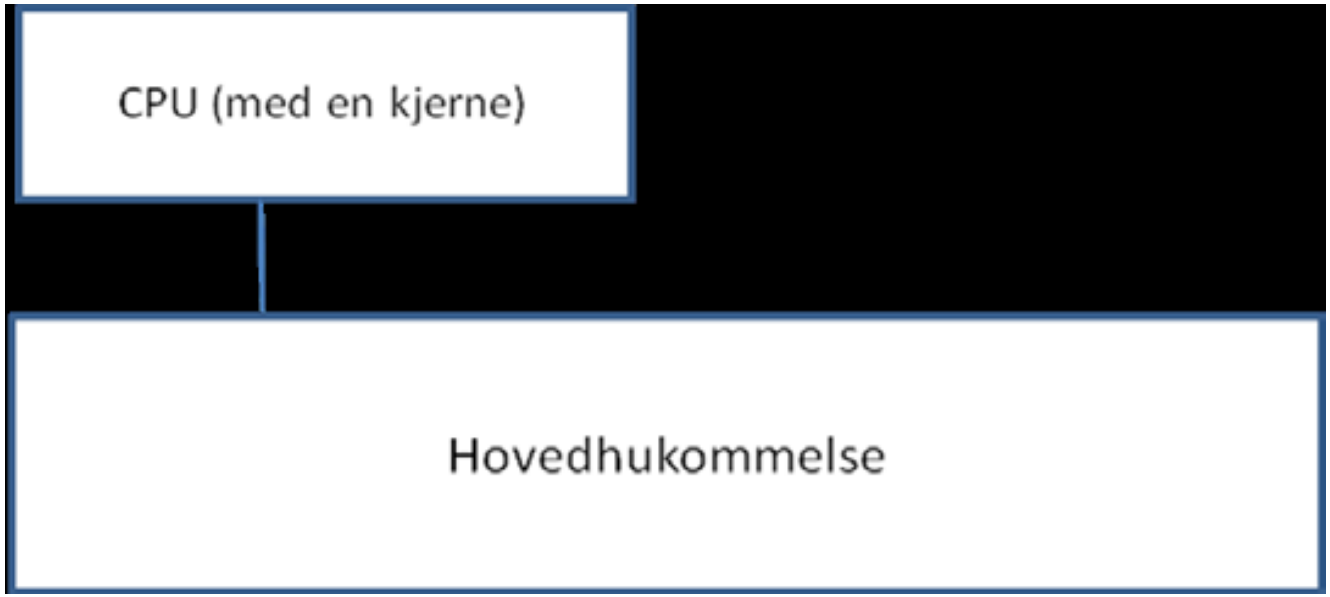


- PRAM (Parallel Random Access Memory) antar to ting:
  - Du har uendelig mange kjerner til beregningene
  - Enhver aksess i lageret tar like lag tid,
    - ignorerer f.eks fordelen med cache-hukommelsen
- Da blir mange algoritmer lette å beregne og programmere
- Problemet er at begge forutsetningene er helt gale.
- Det PRAM gjør er å telle antall instruksjoner utført

  Det har vi sett er helt feilaktig (Radix og Matrise-mult)
- Svært mange kurs og lærebøker er basert på PRAM

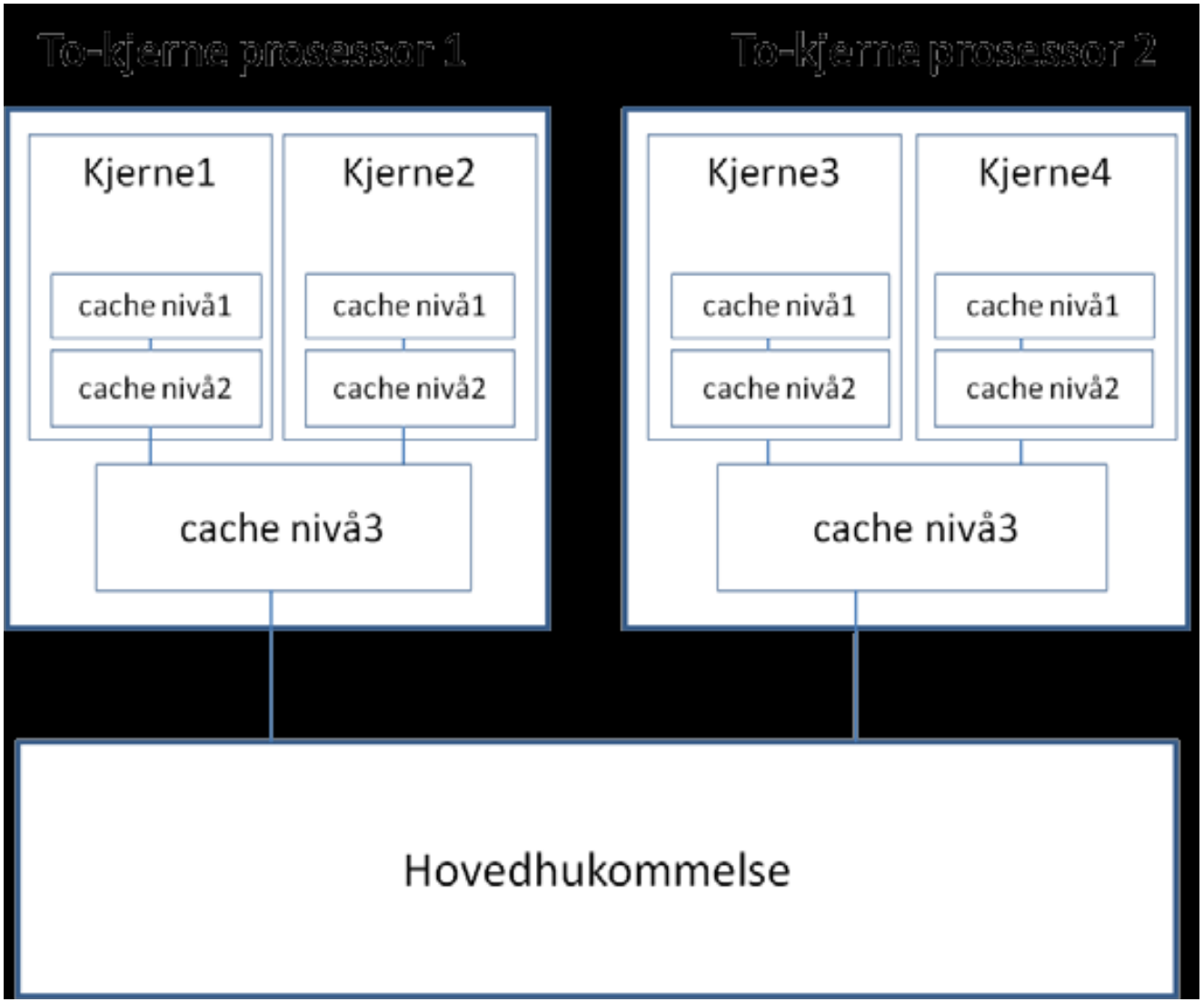
- PRAM vil si at de to sekvensielle algoritmene (med og uten transponering) gikk den utransponerte fortest!
- Dette kurset bruker **ikke** PRAM-modellen!

Maskin 1980 (uten cache)

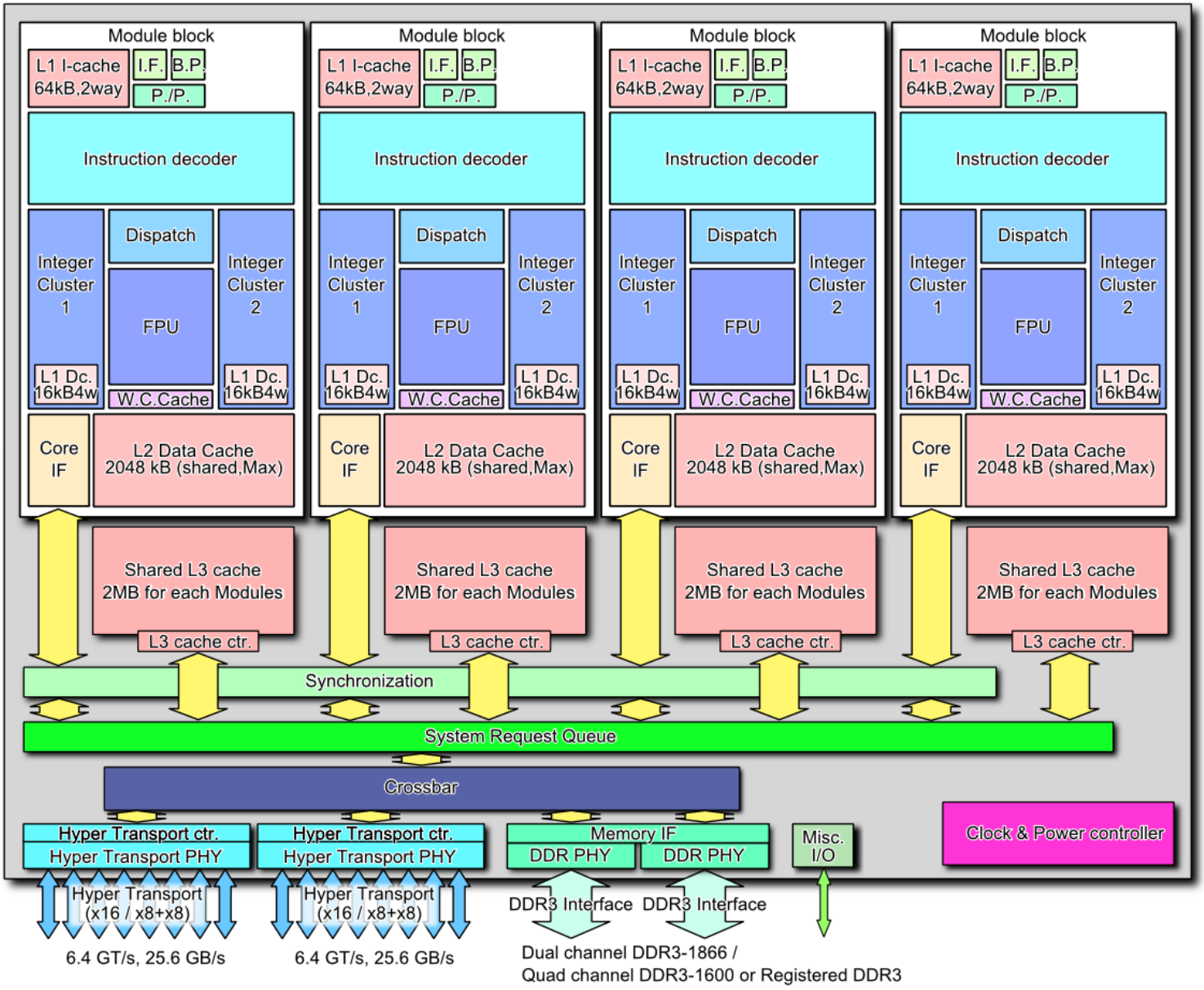


Figur 19.1 Skisse av en datamaskin i ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variable) direkte i hovedhukommelsen. Intel 8080: 1 MHz CPU

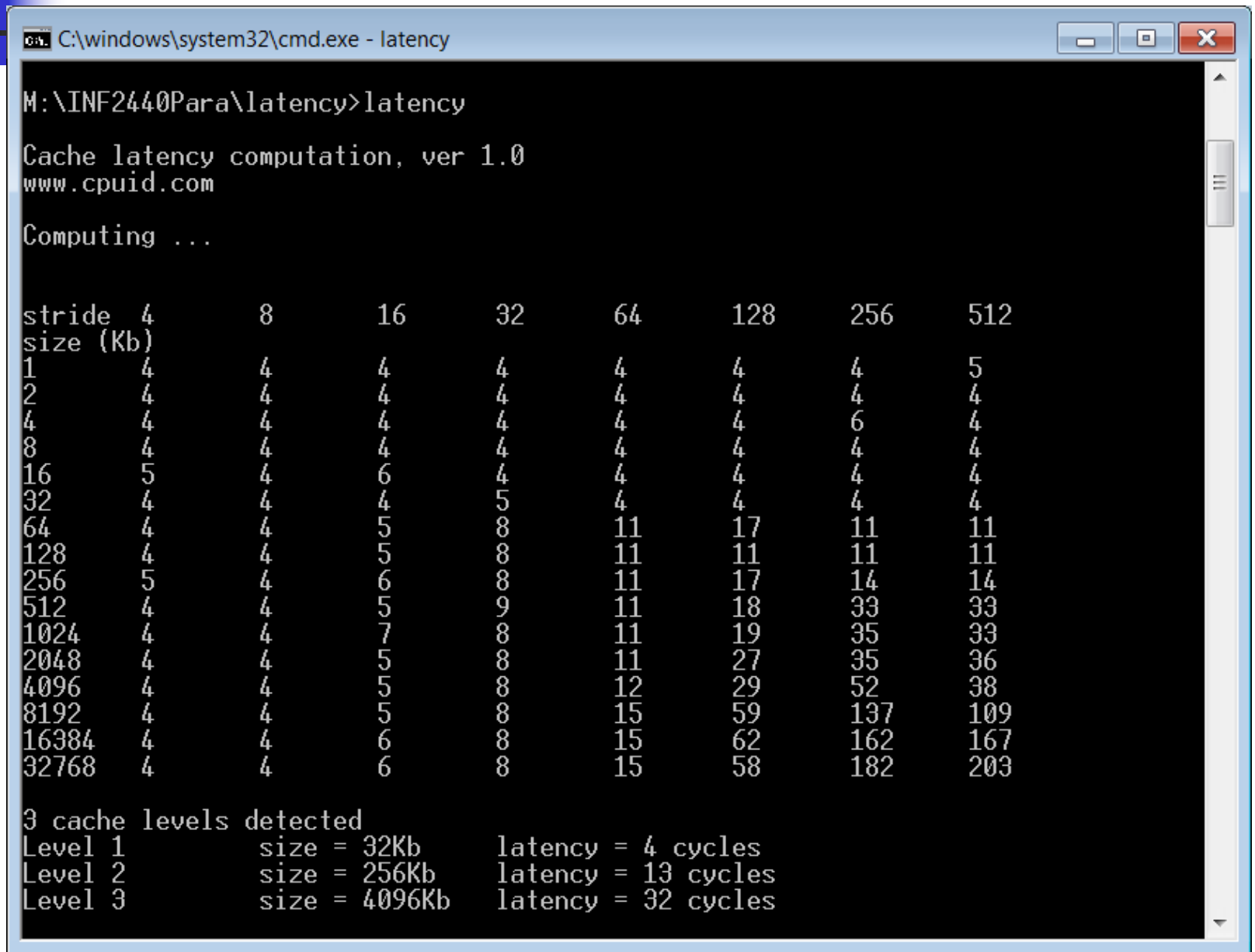
Maskin ca. 2010 med to dobbeltkjerne CPU-er



Hukommelses-systemet i en 4 kjerne CPU – mange lag og flere ulike beregningsmoduler i hver kjerne.:



Test av forsinkelse i data-cachene og hovedhukommelsen - latency.exe (fra CPUZ)

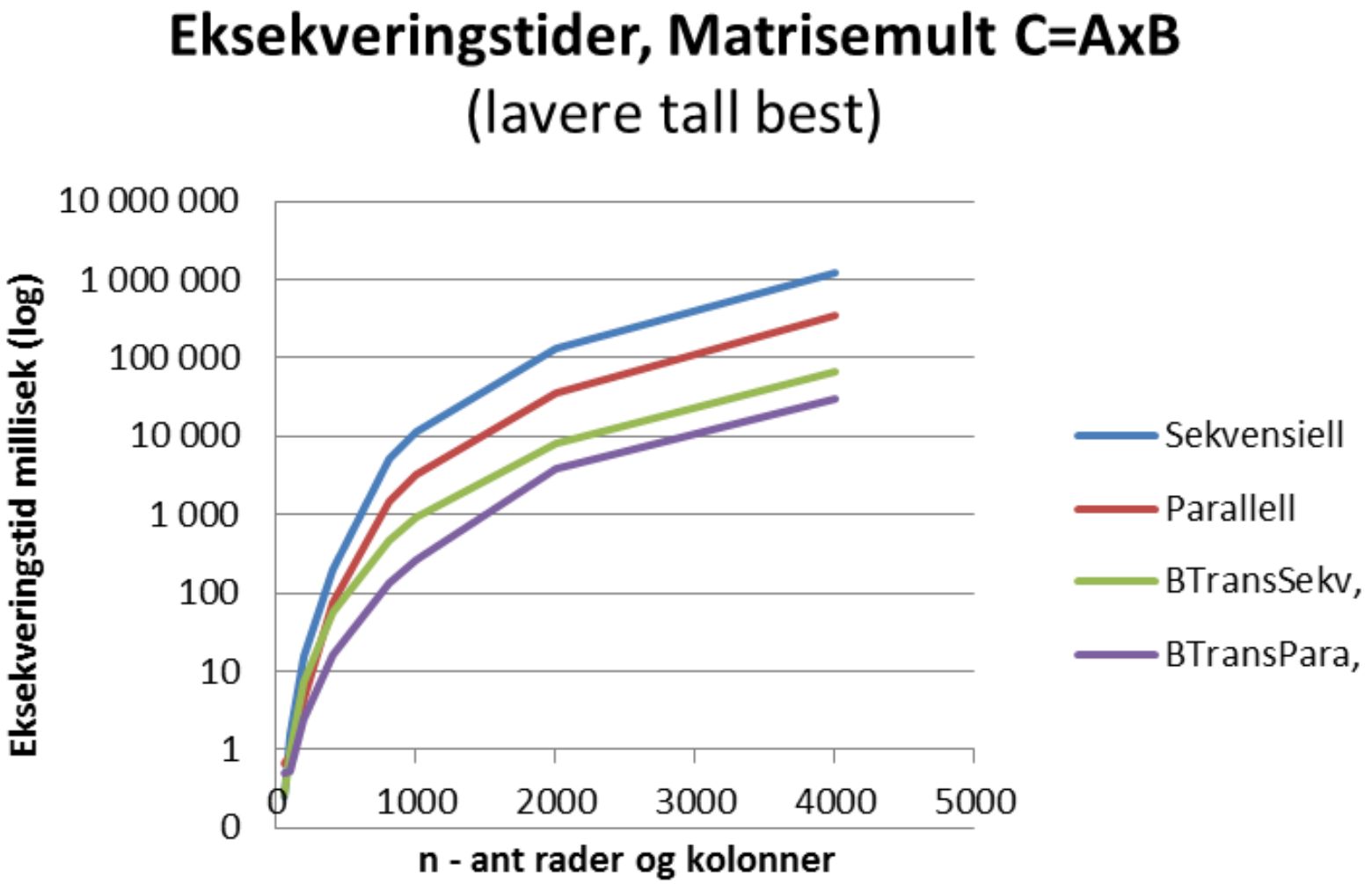


Oppsummering – ideen om at vi har *uniform* aksesstid i hukommelsen er helt galt

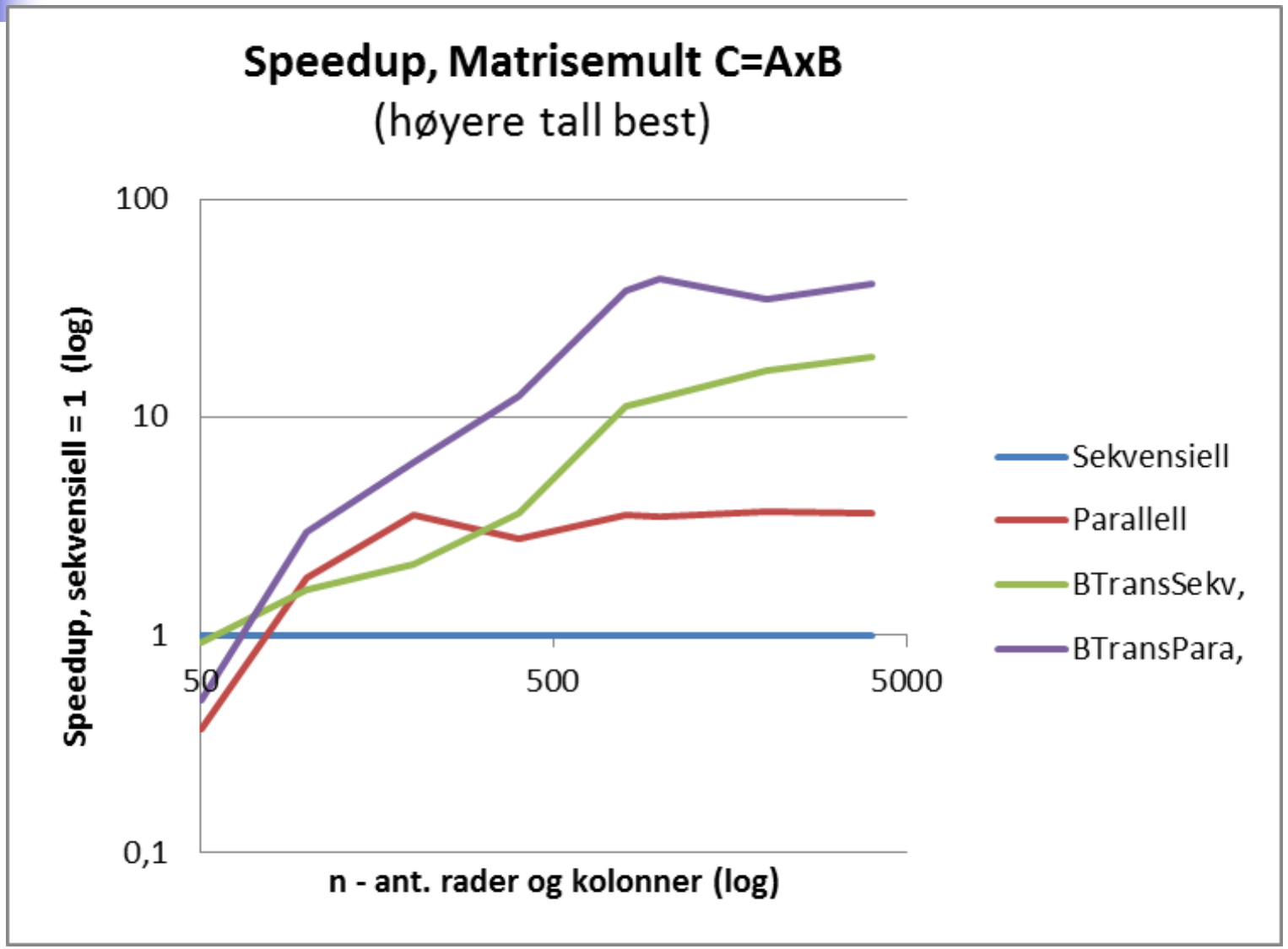
- Hukommelses-systemet i en multicore CPU ,Intel Core i5-459 3.3 GHz, – mange lag (typisk aksesstid i instruksjonssykler):

1. Registre i kjernen (1) – 8/32 registre
2. L1 cache (3-4) – 32 Kb
3. L2 cache (13) – 256 kb
4. L3 cache (32) – 8Mb
5. Hovedhukommelsen (virtuell hukommelse) (ca. 200) – 8-64 GB
6. Disken (15 000 000 roterende) = 5 ms – 1000 GB – 1-5 TB
   FlashDisk (ca 2 000 000 les, ca. 10 000 000 skriv) = ca. 1 ms

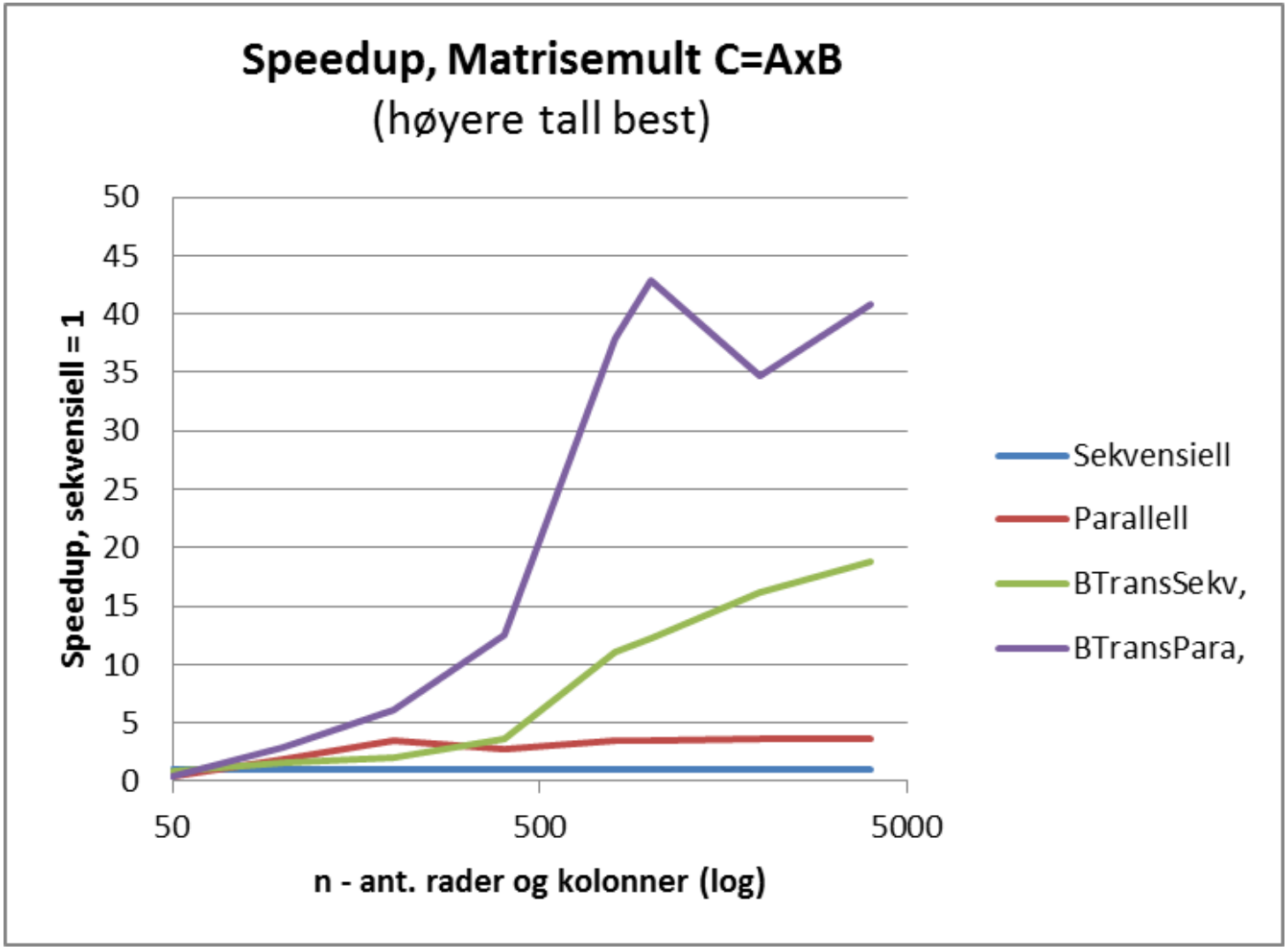
Kjøretider – i millisek. (y-aksen logaritmisk)



Kjøretidsresultater – Speedup , y-aksen logaritmisk

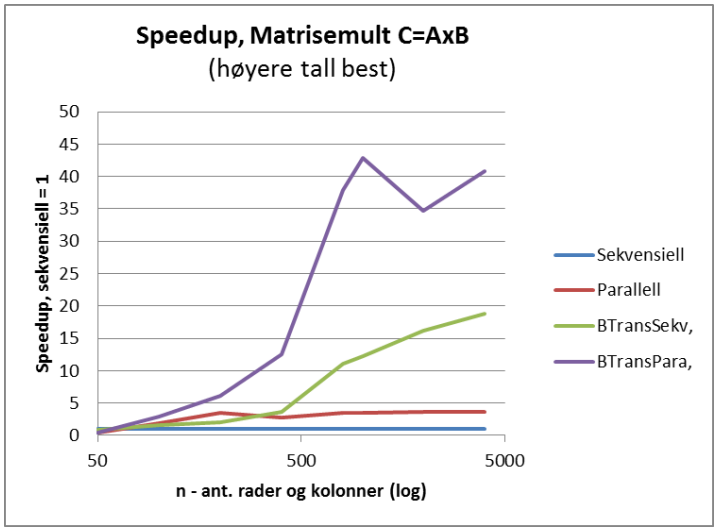


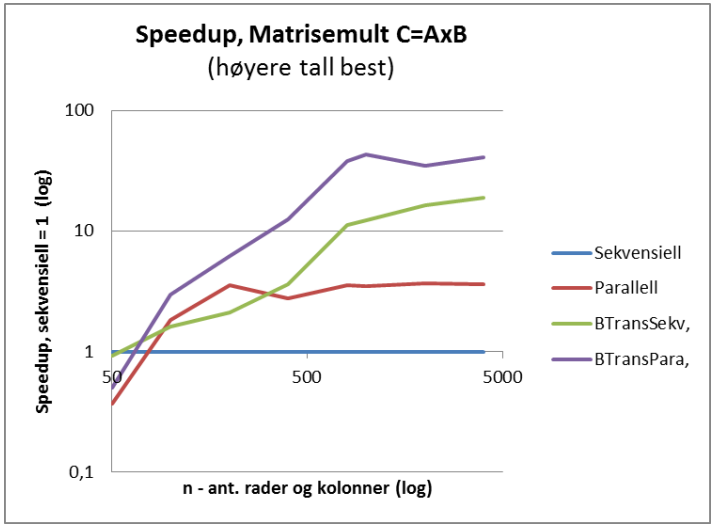
Speedup – med lineær y-akse

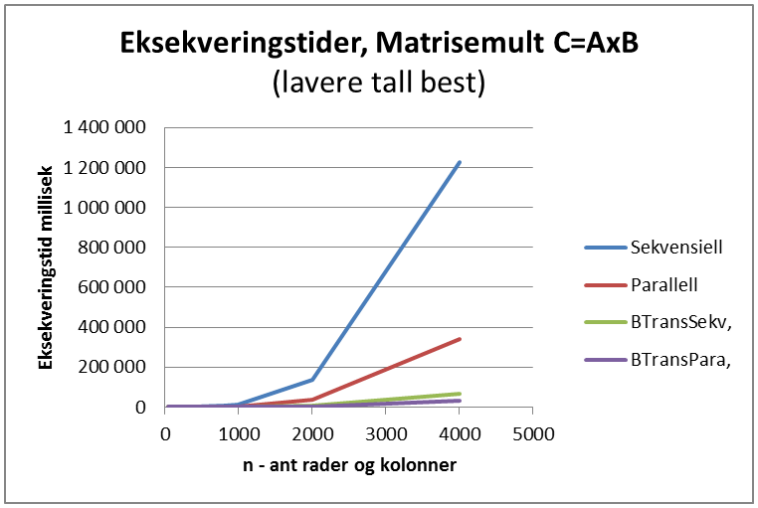


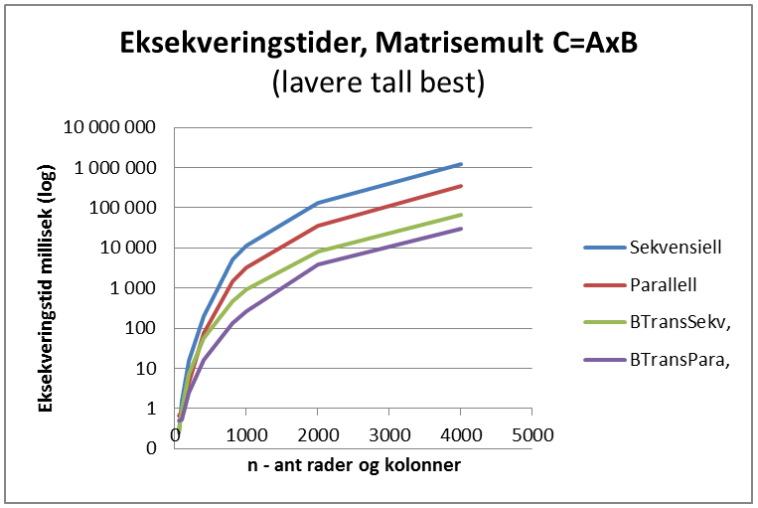
Hvordan framstille ytelse I

- Disse fire kurvene fremviser samme tall! Hvordan ?

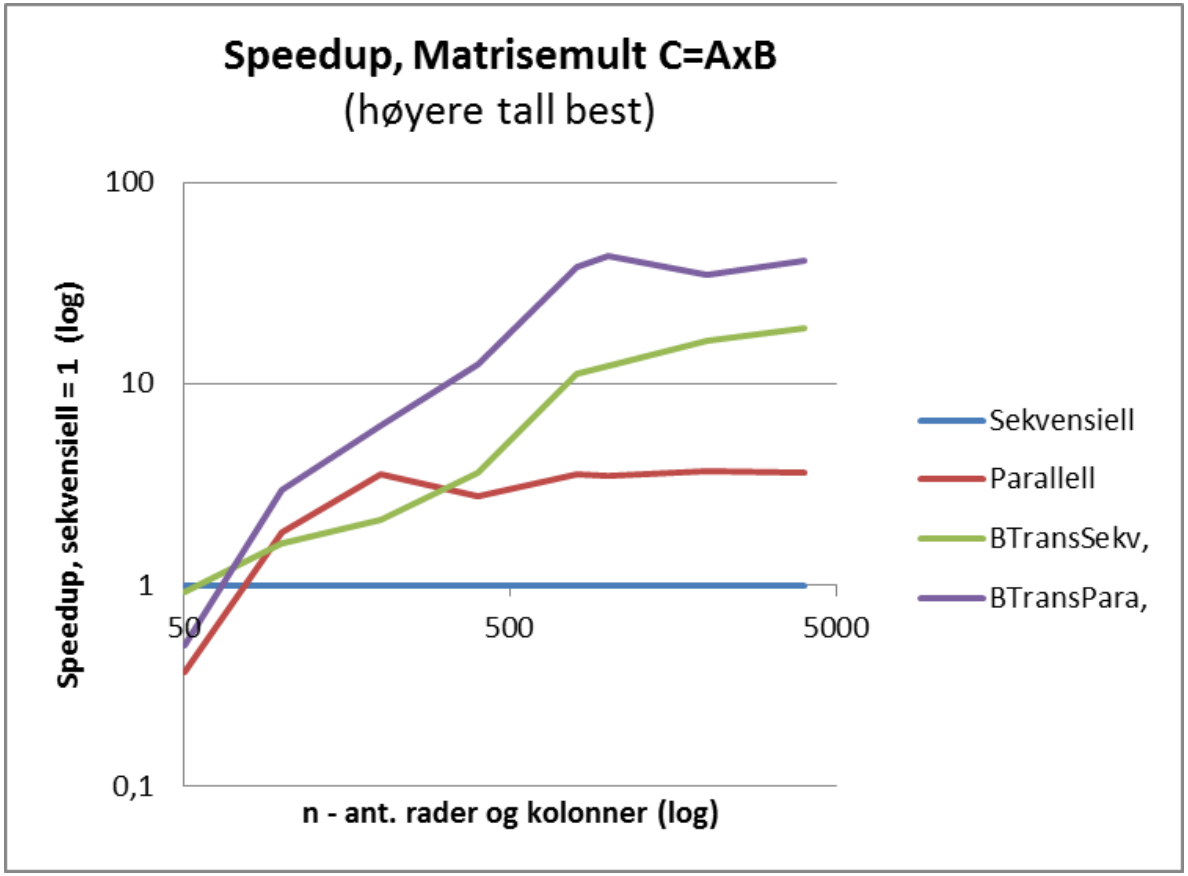




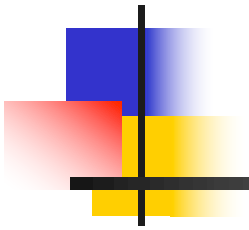




Både logaritmisk x- og y-akse

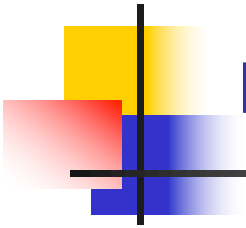


Fordel med log-akse er at den viser fram nøyaktigere små verdier, men vanskelig å lese nøyaktig mellom to merker på aksene.



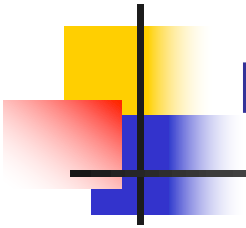
IN3030 L06v23 – Amdahl, Gustavson, Prime Numbers

Eric Jul
Programming Technology Group
Department of Informatics
University of Oslo



Review F05

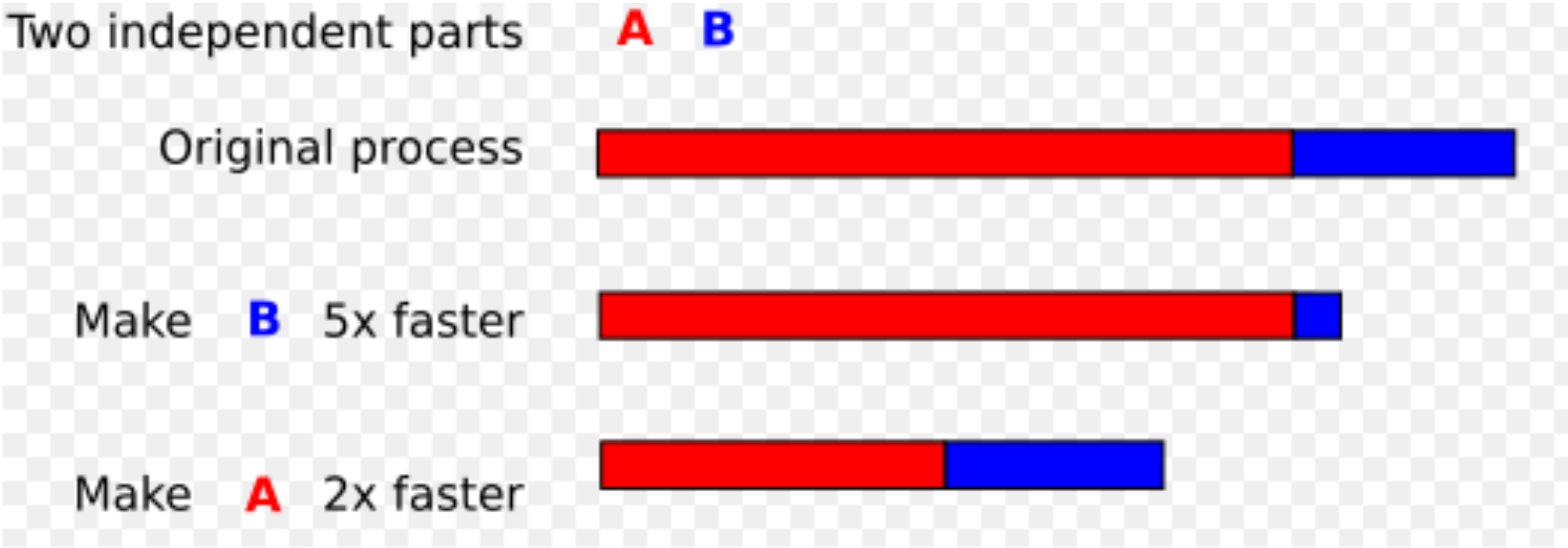
I. Moore's Law
II. Speed of light
III. Latency caused by the slow speed of light
IV. Why distribution
V. How to present your timing results
VI. Hva er PRAM modellen - og hvorfor er den ubrukelig for oss



Plan for F06

I. GPU Exercises this week
II. Amdahl's and Gustavson's laws
III. Prime Numbers
IV. Tidtagning
    I. JIT compilation
    II. Operativsystem?
    III. Søppel/Garbage Collection

Amdahl – viktig å parallellisere største del



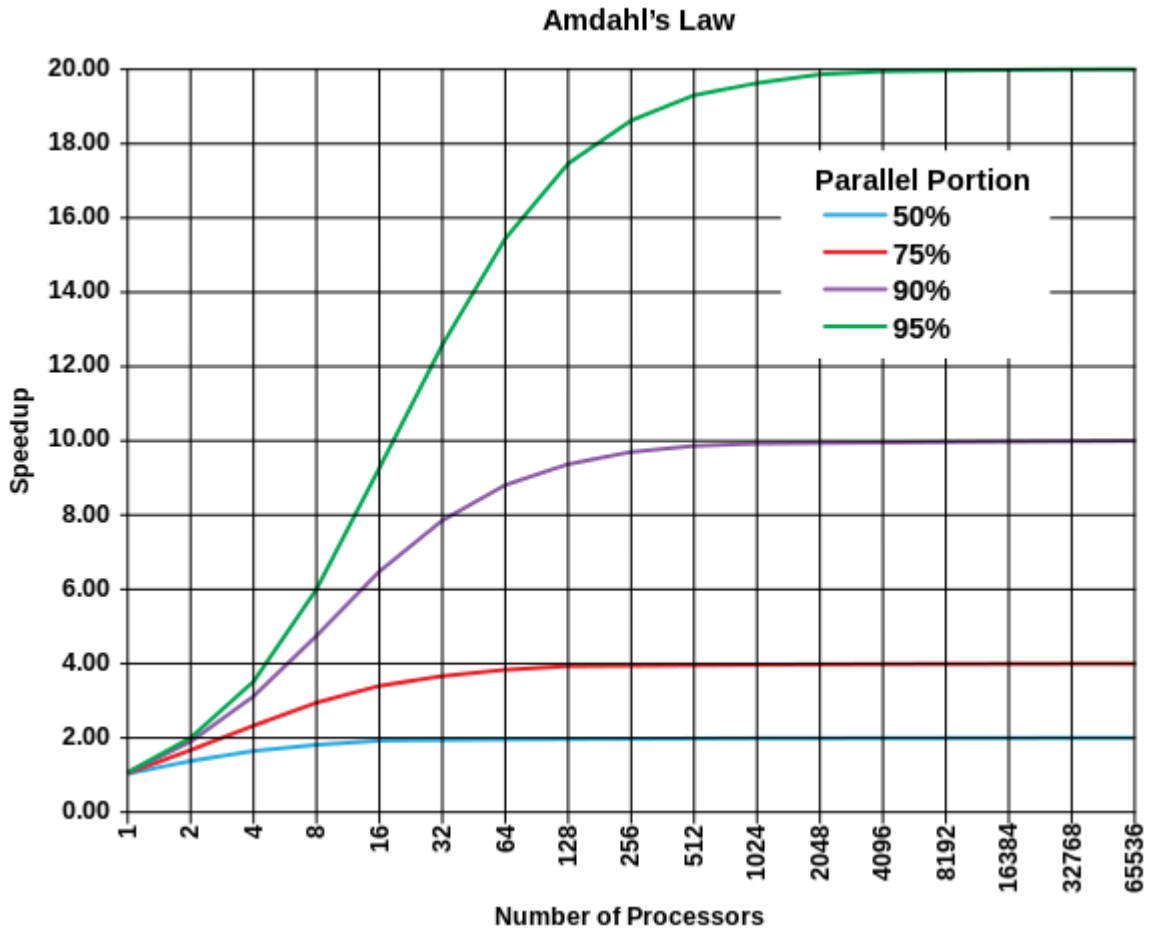
Amdahl lov for parallelle beregninger

- Amdahl lov: Har du **seq** andel sekvensiell kode og da **p** andel parallelliserbar kode i et parallelt program, **seq+p=1**, er den største speedup S du kan få med k kjerner:

$$S = \frac{tid(sekvensiell)}{tid\ (parallell)} = \frac{1}{seq+p/k} = \frac{1}{1-p+p/k}$$

- Når k $\rightarrow\ \infty$, vil S $\rightarrow \frac{1}{1-p}$.
- Er p=0.9, så er S ≤ 10 uansett hvor mange kjerner du har, og har du 'bare' 50, er S = $\frac{1}{1-0.9+0,9/50}$ = 8,5.
- Amdahls lov er pessimistisk- antar fast størrelse på problemet
- «Hvis du først har brukt 10% av tida på en sekvensiell del, så kan resten av programmet ikke gå fortere enn 0.00 sekunder uansett hvor mange prosessorer du bruker på det. Dvs. at speedup ≤ 10»

Amdahl for ulike verdier av p



Gustafsons lov for parallelle beregninger

- La S være speedup, P antall kjerner og α andel sekvensiell kode, så er:

$$S(P) = P - \alpha\,(P-1)$$

Fordi:

Parallell løsning: $a + b\ (a = sekvensiell\ tid, b = parallel\ tid)$

Sekvensiell løsning : $a + P * b$

Speedup er da:

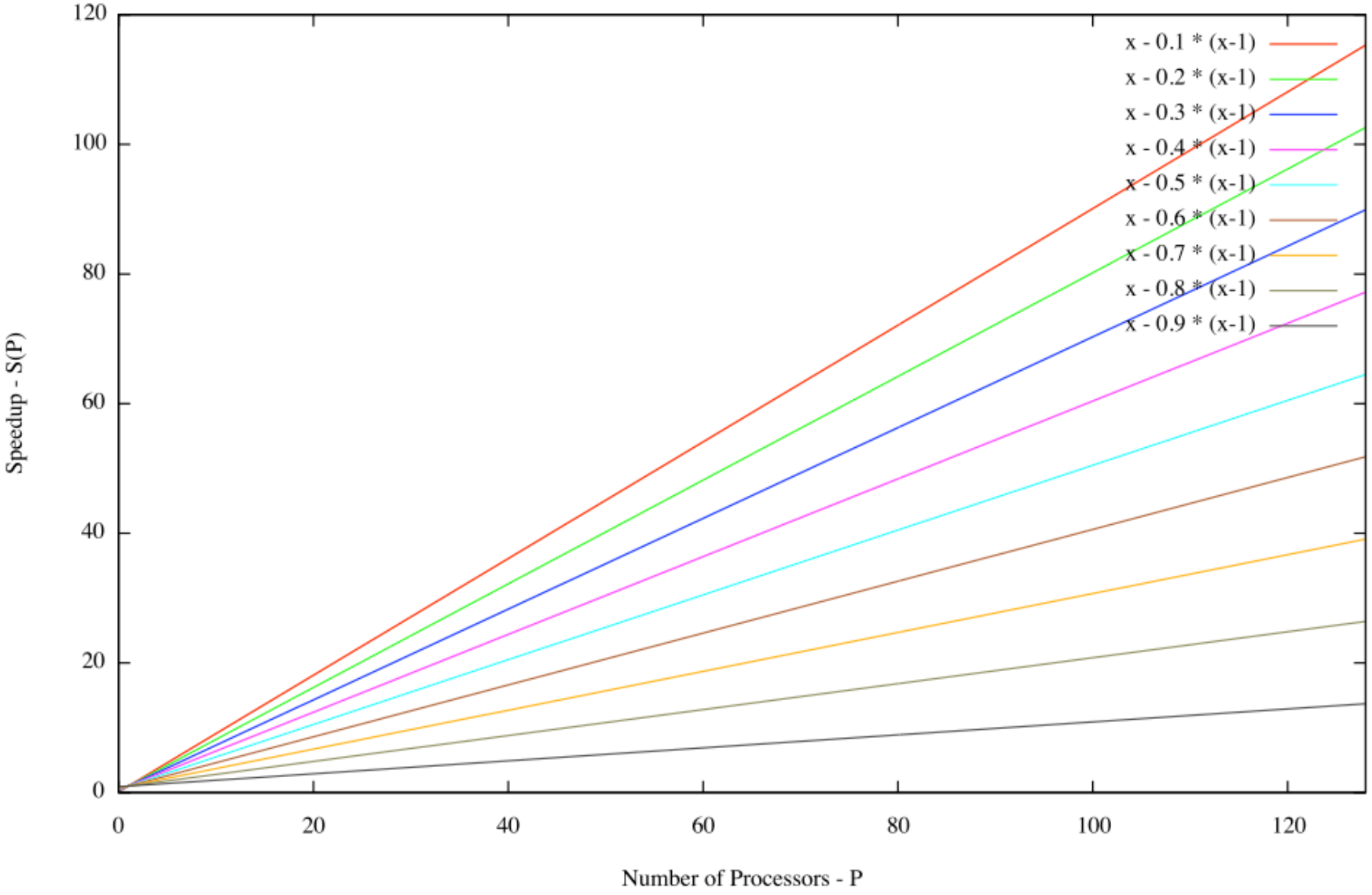
$(a + P * b)/(a + b\ )$, og definerer $\alpha = \frac{a}{a+b}$, så er:

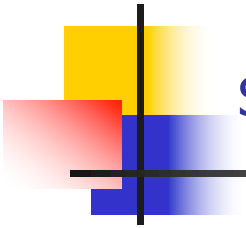
$$S(P) = \alpha + P * (1 - \alpha) = P\ - \alpha(P-1)$$

- Gustavson er mer optimistisk enn Amdahl, gir høyere speedup fordi han antar at med flere maskiner vil vi øke størrelsen på problemet.

- «Hvis du tidligere brukte 1 time på å løse et problem sekvensielt, vil du nå også bruke 1 time på å løse et større, mer nøyaktig problem parallelt da med større speedup– for eksempel i meteorologi»

Gustafson's Law: $S(x) = x - \alpha(x - 1),$



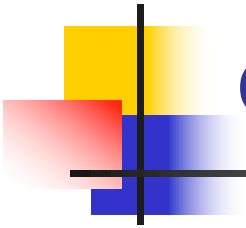


Sammenligning av Amdahl og Gustafson + egne betraktninger

- Amdahl antar at oppgaven er fast av en gitt lengde(n)
- Gustafson antar at du med parallelle maskiner løser større problemer (større n) og da blir den sekvensielle delen mindre.
- Min betraktning:
  1. En algoritme består av noen sekvensielle deler og noen parallelliserbare deler.
  2. Hvis de sekvensielle delene har lavere orden – f.eks O(log n), men de parallelle har en større orden – eks O(n) så vil de parallelle delene bli en stadig større del av kjøretida hvis n øker (Gustafson)
  3. Hvis de parallelle og sekvensielle delene har samme orden, vil et større problem ha samme sekvensielle andel som et mindre problem (Amdahl).
  4. I tillegg kommer alltid et fast overhead på å starte k tråder (1-4 ms.)

  Algoritmer vi skal jobbe med er mer av type 2 (Gustafson) enn type 3(Amdahl) men vi har alltid overhead, så små problemer løses best sekvensielt.

Konklusjon: For store problemer bør vi ha håp om å skalere nær lineært med antall kjerner hvis ikke vi får kø og forsinkelser når alle kjernene skal lese/skrive i lageret.



Om primtall

- Primtall og faktorisering av ikke-primtall.
- Et primtall er:
  Et heltall som bare lar seg dividere med 1 og seg selv.
  - 1 er ikke et primtall (det mente mange på 1700-tallet, og noen mener det fortsatt)

Om primtall og faktorisering av heltall

- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall

2 måter å lage primtall

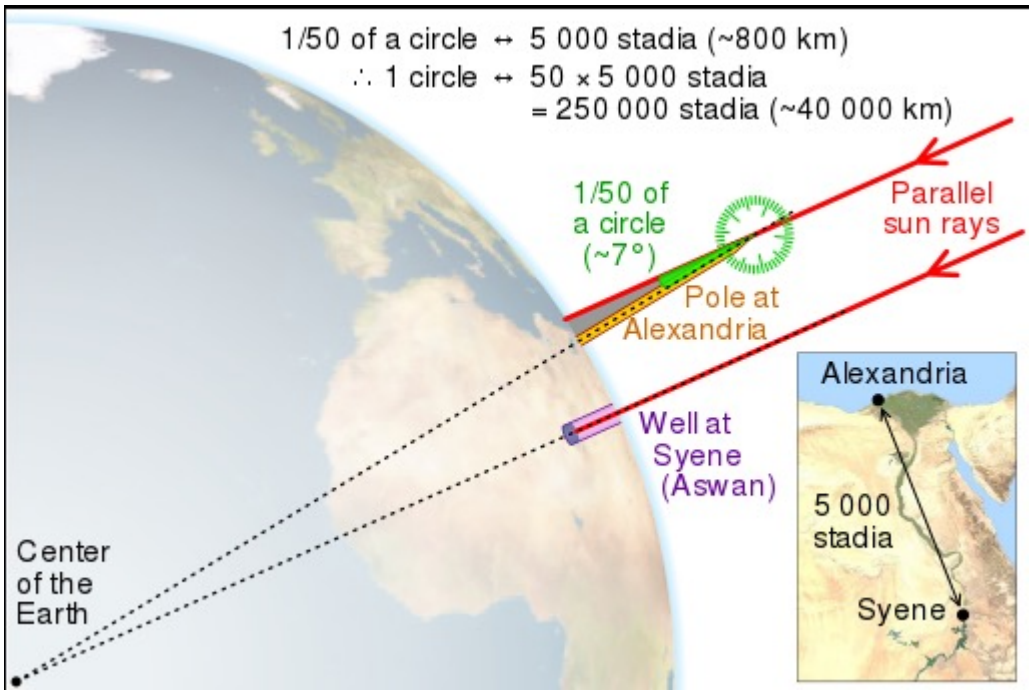
Ønsker at finne alle primtal $p_i < N$

- Dividere alle tall < N med alle tall < N
  - Divisjonsmetoden
  - Bare oddetall (2 spesiel)
  - Bare opp til $\sqrt{N}$ -- hvorfor?
  - Bare primtall opp til $\sqrt{N}$ -- hvorfor?

- Lage en tabell over alle de primtallene vi trenger
  - Eratosthene sil

Litt mer om Eratosthenes

Eratosthenes, matematikker, laget også et estimat på jordas radius som var < 1,5% feil, grunnla geografi som fag, fant opp skuddårsdagen + at han var sjef for Biblioteket i Alexandria (den tids største forskningsinstitusjon).



Hvad er raskest?

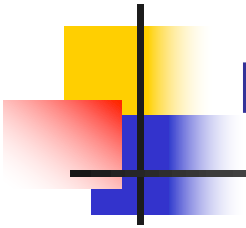
- A) Med Eratosthenes sil:

```
Z:\INF2440Para\Primtall>java PrimtallESil 2000000000
 max primtall m:2000000000
Genererte alle primtall <= 2000000000  paa     18 949 millisek
med Eratosthenes sil  og det største primtallet er:1999999973
```

- Med gjentatte divisjoner

```
Z:\INF2440Para\Primtall>java PrimtallDiv 2000000000
Genererte alle primtall <=2000000000  paa   1 577 302  millisek med
 divisjon , og det største primtallet er:1999999973
```

- Å lage primtallene p og finne dem ved divisjon (del på alle oddetall < SQRT(p), p = 3,5,7,..) er ca. 100 ganger langsommere enn Eratosthenes avkryssings-tabell (kalt Eratosthenes sil).



Finne primtall -- Eratosthenes sil

- Hvordan?
- (Blackboard)

Om primtall og faktorisering av heltall

- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall
- Eksempel: faktorisering av 532

Å lage og lagre primtall (Erotosthenes sil)

- Som en bit-tabell (1- betyr primtall, 0-betyr ikke-primtall)
  - Påfunnet i jernalderen av Eratosthenes (ca. 200 f.kr)
  - Man skal finne alle primtall < M
  - Man finner da de første primtallene og krysser av alle multipla av disse (N.B. dette forbedres/endres senere):
    - Eks: 3 er et primtall, da krysses 6, 9,12,15,.. Av fordi de alle er ett-eller-annet-tall (1,2,3,4,5,..) ganger 3 og følgelig selv ikke er et primtall. 6=2*3, 9 = 3*3,
      12 =2*2*3, 15 = 5*3, ..osv
    - De tallene som *ikke blir* krysset av, når vi har krysset av for alle primtallene vi har, er primtallene
- Vi finner 5 som et primtall fordi, etter at vi har krysset av for 3, finner første ikke-avkryssete tall: 5, som da er et primtall (og som vi så krysser av for, …finner så 7 osv)

Litt mer om Eratothenes sil

- Vi representerer ikke partallene på den tallinja som det krysses av på fordi vi vet at 2 er et primtall (det første) og at alle andre partall er ikke-primtall.
- Har vi funnet et nytt primtall p, for eksempel. 5, starter vi avkryssingen for dette primtallet først for tallet p*p (i eksempelet: 25), men etter det krysses det av for p*p+2p, p*p+4p,.. (i eksempelet 35,45,55,…osv.). Grunnen til at vi kan starte på p*p er at alle andre tall t < p*p slik det krysses av i for eksempel Wikipedia-artikkelen har allerede blitt krysset av andre primtall < p.
- Det betyr at for å krysse av og finne alle primtall < N , behøver vi bare å krysse av på denne måten for alle primtall p ≤ sqrt(N). Dette sparer svært mye tid.

Vise at vi trenger bare primtallene <10 for å finne alle primtall < 100, avkryssing for 3 (3*3, 9+2*3,9+4*3, ….)

1	**3**	**5**	**7**	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	**3**	**5**	**7**	**9**
11	13	**15**	17	19
21	23	25	**27**	29
31	**33**	35	37	**39**
41	43	**45**	47	49
51	53	55	**57**	59
61	**63**	65	67	**69**
71	73	**75**	77	79
81	83	85	**87**	89
91	**93**	95	97	**99**

Avkryssing for 5 (starter med 25, så 25+2*5, 25+4,5,..):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

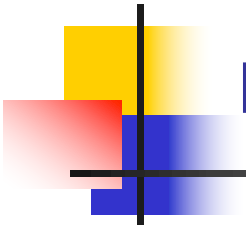
1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Avkryssing for 7 (starter med 49, så 49+2*7,49+4*7,.):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63 63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Er nå ferdig fordi neste primtall vi finner: 11, så er 11*11=121 utenfor tabellen

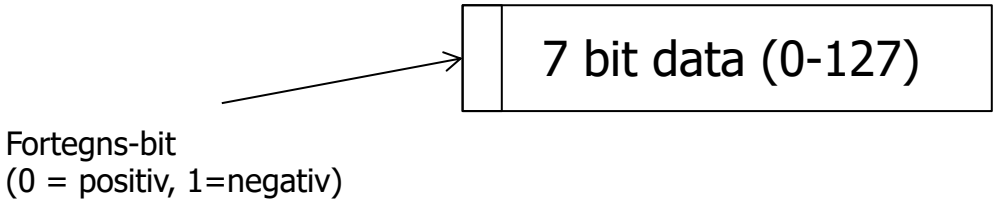


Hvordan representeres tallene?

- Kun oddetall – 2 kjenner vi!
- Array of Boolean?
  - Problem: 32 bit per primtall
- Kompakter bitarray
    - Kun 1 bit per oddetall

Hvordan bruke 8 eller 7 bit i en **byte-array** for å representere primtallene

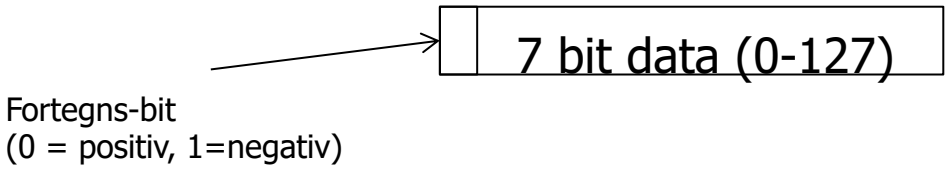
En byte = 8 bit heltall:



- Vi representer alle oddetallene (1,3,5,,,) som ett bit (0= ikke-primtall, 1 = primtall)
- Bruke alle 8 bit :
  - Fordel: mer kompakt lagring og litt raskere(?) adressering
  - Ulempe: Kan da ikke bruke verdien i byten direkte (f.eks som en indeks til en array), heller ikke +,-,* eller /-operasjonene på verdien
- Bruke 7 bit:
  - Fordel: ingen av ulempene med 8 bit
  - Ulempe: Tar litt større plass og litt langsommere(?) adressering

Hvordan representere 8 (eller 7) bit i en byte-array

byte = et 8 bit heltall



- Bruker alle 8 bitene til oddetallene:
  - Anta at vi vil sjekke om tallet ***k*** er et primtall, sjekk først om ***k*** er 2, da ja, hvis det er et partall (men ikke 2) da nei – ellers sjekk så tallets bit i byte-arrayen
    - Byte nummeret til ***k*** i arrayen er da:
      - Enten: ***k***/16, eller: ***k*** >>>4 (shift 4 høyreover uten kopi av fortegns-bitet er det samme som å dele med 16)
    - Bit-nummeret er i denne byten er da enten (***k*** %16)/2 eller (***k*** &15)>>1
  - Hvorfor dele på 16 når det er 8 bit
    - fordi vi fjernet alle partallene – egentlig 16 tall representert i første byten, for byte 0: tallene 0-15
  - Om så å finne bitverdien – se neste lysark.

Bruke 7 bit i hver byte i arrayen

- Anta at vi vil sjekke om tallet ***k*** er et primtall sjekk først om ***k*** er 2, da ja, ellers hvis det er et partall (men ikke 2) da nei – ellers:
- Sjekk da tallets bit i byte-arrayen
  - Byte nummeret til ***k*** i arrayen er da: ***k***/14
  - Bit-nummeret er i denne byten er da: (***k*** %14)/2
- Nå har vi byte-nummeret og bit-nummeret i den byten. Vi kan da ta AND (&) med det riktige elementet i en av de to arrayene som er oppgitt i skjelett-koden og teste om svaret er 0 eller ikke.
- Hvordan sette alle 7 eller 8 bit == 1 i alle byter)
  - 7 bit: hver byte settes = 127 (men bitet for 1 settes =0)
  - 8 bit: hver byte settes = -1 (men bit for 1 settes = 0)

- Konklusjon: bruk 8 eller 7 bit i hver byte (valgfritt) i Oblig3

Faktorisering av et tall M i sine primtallsfaktorer

- Vi har laget og lagret ved hjelp av Erotosthanes sil alle (unntatt 2) primtall < N i en bit-array over alle odde-tallene.
  - 1 = primtall, 0=ikke-primtall
  - Vi har krysset ut de som ikke er primtall
- Hvordan skal vi så bruke dette til å faktorisere et tall M < N*N ?
- **Svar:** Divider M med alle primtall $p_i < \sqrt{M}$ (p_i =2,3,5,..), og hver gang en slik divisjon M%p_i ==0, så er p_i en av faktorene til M. Vi forsetter så med å faktorisere ett mindre tall M'=M/p_i.
- Faktoriseringen av M = p_i *..* p_k er da produktet av alle de primtall som dividerer M uten rest.
- HUSK at en p_i kan forekommer flere ganger i svaret. eks: 20= 2*2*5, 81 = 3*3*3*3, osv
- Finner vi ingen faktorisering av M, dvs. ingen $p_i \leq \sqrt{M}$ som dividerer M med rest == 0, så er M selv et primtall.

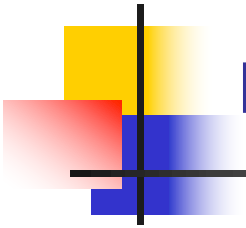
Hvordan parallellisere faktorisering ?

1. Gjennomgås neste uke - denne uka viktig å få på plass en effektiv sekvensiell løsning med om lag disse kjøretidene for N = 2 mill:

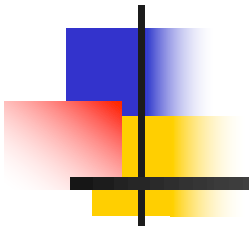
```
M:\INF2440Para\Primtall>java PrimtallESil 2000000
 max primtall m:2 000 000
Genererte primtall <= 2000000  paa        15.56 millisek
med Eratosthenes sil (  0.00004182 millisek/primtall)
.....
3999998764380 = 2*2*3*5*103*647248991
3999998764381 = 37*108108074713
3999998764382 = 2*271*457*1931*8363
3999998764383 = 3*19*47*1493093977
3999998764384 = 2*2*2*2*2*7*313*1033*55229
3999998764385 = 5*13*59951*1026479
3999998764386 = 2*3*3*31*71*100964177
3999998764387 = 1163*1879*1830431
3999998764388 = 2*2*11*11*17*23*293*72139
 100 faktoriseringer beregnet paa:     422.0307ms -
 dvs:      4.2203ms. per faktorisering
```

Faktorisering av store tall med 18-19 desimale sifre

```
Uke5>java PrimtallESil 2140000000
max primtall m:2 140 000 000
bitArr.length:133 750 001
Genererte primtall <= 2 140 000 000  paa    11030.36 millisek
med Eratosthenes sil ( 0.00010530 millisec/primtall)
 antall primtall < 2 140 000 000 er: 104 748 779, dvs:  4.89% ,
 og det største primtallet er: 2 139 999 977

4 579 599 999 999 999 900 = 2*2*3*5*5*967*3673*19421*221303
4 579 599 999 999 999 901 = 45795999999999901
4 579 599 999 999 999 902 = 2*22897999999999951
4 579 599 999 999 999 903 = 3*31*13188589*3733758839
4 579 599 999 999 999 904 = 2*2*2*2*2*19*71*106087842846553
4 579 599 999 999 999 905 = 5*7*1308457142857142 83
.............
4 579 599 999 999 999 997 = 11*4163272727272727
4 579 599 999 999 999 998 = 2*121081*18911307306679
4 579 599 999 999 999 999 = 3*17*19*6625387*713333333
100 faktoriseringer beregnet paa:  333481.4427ms
 dvs:   3334.8144ms. per faktorisering

 largestLongFactorizedSafe: 4 579 599 841 640 001 173= 2139999949*2139999977
```

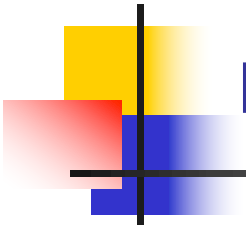


End of lecture L06v23



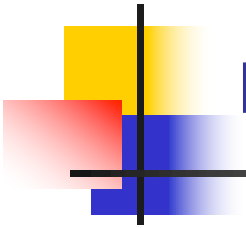
IN3030 L07v23 – Prime Numbers, Timing

Eric Jul
Programming Technology Group
Department of Informatics
University of Oslo
2023-03-08



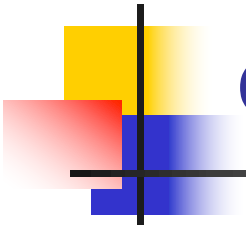
Review L06v23

I. GPU Exercises this week
II. Amdahl's and Gustavson's laws
III. Prime Numbers, Erasthophenes sieve



Plan for L07v23

I. Prime Numbers, review
II. Oblig 3: Prime Numbers
III. Tidtagning
    I. JIT compilation
    II. Operativsystem?
    III. Søppel/Garbage Collection



Om primtall

- Primtall og faktorisering av ikke-primtall.
- Et primtall er:
  Et heltall som bare lar seg dividere med 1 og seg selv.
  - 1 er ikke et primtall (det mente mange på 1700-tallet, og noen mener det fortsatt)

Om primtall og faktorisering av heltall

- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall

2 måter å lage primtall

Ønsker at finne alle primtal $p_i < N$

- Dividere alle tall < N med alle tall < N
  - Divisjonsmetoden
  - Bare oddetall (2 spesiel)
  - Bare opp til $\sqrt{N}$ -- hvorfor?
  - Bare primtall opp til $\sqrt{N}$ -- hvorfor?

- Lage en tabell over alle de primtallene vi trenger
  - Eratosthene sil

Hvad er raskest?

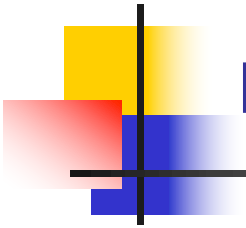
- A) Med Eratosthenes sil:

```
Z:\INF2440Para\Primtall>java PrimtallESil 2000000000
 max primtall m:2000000000
Genererte alle primtall <= 2000000000  paa     18 949 millisek
med Eratosthenes sil  og det største primtallet er:1999999973
```

- Med gjentatte divisjoner

```
Z:\INF2440Para\Primtall>java PrimtallDiv 2000000000
Genererte alle primtall <=2000000000  paa   1 577 302  millisek med
 divisjon , og det største primtallet er:1999999973
```

- Å lage primtallene p og finne dem ved divisjon (del på alle oddetall < SQRT(p), p = 3,5,7,..) er ca. 100 ganger langsommere enn Eratosthenes avkryssings-tabell (kalt Eratosthenes sil).



Finne primtall -- Eratosthenes sil

- Hvordan?
- (Blackboard)

Å lage og lagre primtall (Erotosthenes sil)

- Som en bit-tabell (1- betyr primtall, 0-betyr ikke-primtall)
  - Påfunnet i jernalderen av Eratosthenes (ca. 200 f.kr)
  - Man skal finne alle primtall < M
  - Man finner da de første primtallene og krysser av alle multipla av disse (N.B. dette forbedres/endres senere):
    - Eks: 3 er et primtall, da krysses 6, 9,12,15,.. Av fordi de alle er ett-eller-annet-tall (1,2,3,4,5,..) ganger 3 og følgelig selv ikke er et primtall. 6=2*3, 9 = 3*3,
      12 =2*2*3, 15 = 5*3, ..osv
    - De tallene som *ikke blir* krysset av, når vi har krysset av for alle primtallene vi har, er primtallene
- Vi finner 5 som et primtall fordi, etter at vi har krysset av for 3, finner første ikke-avkryssete tall: 5, som da er et primtall (og som vi så krysser av for, …finner så 7 osv)

Litt mer om Eratothenes sil

- Vi representerer ikke partallene på den tallinja som det krysses av på fordi vi vet at 2 er et primtall (det første) og at alle andre partall er ikke-primtall.
- Har vi funnet et nytt primtall p, for eksempel. 5, starter vi avkryssingen for dette primtallet først for tallet p*p (i eksempelet: 25), men etter det krysses det av for p*p+2p, p*p+4p,.. (i eksempelet 35,45,55,…osv.). Grunnen til at vi kan starte på p*p er at alle andre tall t < p*p slik det krysses av i for eksempel Wikipedia-artikkelen har allerede blitt krysset av andre primtall < p.
- Det betyr at for å krysse av og finne alle primtall < N , behøver vi bare å krysse av på denne måten for alle primtall p ≤ sqrt(N). Dette sparer svært mye tid.

Vise at vi trenger bare primtallene <10 for å finne alle primtall < 100, avkryssing for 3 (3*3, 9+2*3,9+4*3, ….)

1	**3**	**5**	**7**	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	**3**	**5**	**7**	**9**
11	13	**15**	17	19
21	23	25	**27**	29
31	**33**	35	37	**39**
41	43	**45**	47	49
51	53	55	**57**	59
61	**63**	65	67	**69**
71	73	**75**	77	79
81	83	85	**87**	89
91	**93**	95	97	**99**

Avkryssing for 5 (starter med 25, så 25+2*5, 25+4,5,..):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Avkryssing for 7 (starter med 49, så 49+2*7,49+4*7,.):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63 63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Er nå ferdig fordi neste primtall vi finner: 11, så er 11*11=121 utenfor tabellen

Faktorisering av et tall M i sine primtallsfaktorer

- Vi har laget og lagret ved hjelp av Erotosthanes sil alle (unntatt 2) primtall < N i en bit-array over alle odde-tallene.
  - 1 = primtall, 0=ikke-primtall
  - Vi har krysset ut de som ikke er primtall
- Hvordan skal vi så bruke dette til å faktorisere et tall M < N*N ?
- **Svar:** Divider M med alle primtall $p_i < \sqrt{M}$ (p_i =2,3,5,..), og hver gang en slik divisjon M%p_i ==0, så er p_i en av faktorene til M. Vi forsetter så med å faktorisere ett mindre tall M'=M/p_i.
- Faktoriseringen av M = p_i *..* p_k er da produktet av alle de primtall som dividerer M uten rest.
- HUSK at en p_i kan forekommer flere ganger i svaret. eks: 20= 2*2*5, 81 = 3*3*3*3, osv
- Finner vi ingen faktorisering av M, dvs. ingen $p_i \leq \sqrt{M}$ som dividerer M med rest == 0, så er M selv et primtall.

Hvordan parallellisere faktorisering ?

1. Denne uka viktig å få på plass en effektiv sekvensiell løsning med om lag disse kjøretidene for N = 2 mill:

```
M:\INF2440Para\Primtall>java PrimtallESil 2000000
 max primtall m:2 000 000
Genererte primtall <= 2000000  paa        15.56 millisek
med Eratosthenes sil (  0.00004182 millisek/primtall)
.....
3999998764380 = 2*2*3*5*103*647248991
3999998764381 = 37*108108074713
3999998764382 = 2*271*457*1931*8363
3999998764383 = 3*19*47*1493093977
3999998764384 = 2*2*2*2*2*7*313*1033*55229
3999998764385 = 5*13*59951*1026479
3999998764386 = 2*3*3*31*71*100964177
3999998764387 = 1163*1879*1830431
3999998764388 = 2*2*11*11*17*23*293*72139
 100 faktoriseringer beregnet paa:     422.0307ms -
 dvs:      4.2203ms. per faktorisering
```

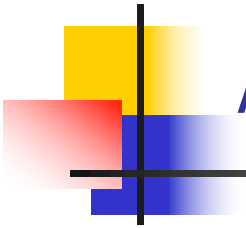
Faktorisering av store tall med 18-19 desimale sifre

```
Uke5>java PrimtallESil 2140000000
max primtall m:2 140 000 000
bitArr.length:133 750 001
Genererte primtall <= 2 140 000 000  paa    11030.36 millisek
med Eratosthenes sil ( 0.00010530 millisec/primtall)
 antall primtall < 2 140 000 000 er: 104 748 779, dvs:  4.89% ,
 og det største primtallet er: 2 139 999 977

4 579 599 999 999 999 900 = 2*2*3*5*5*967*3673*19421*221303
4 579 599 999 999 999 901 = 45795999999999901
4 579 599 999 999 999 902 = 2*22897999999999951
4 579 599 999 999 999 903 = 3*31*13188589*3733758839
4 579 599 999 999 999 904 = 2*2*2*2*2*19*71*106087842846553
4 579 599 999 999 999 905 = 5*7*1308457142857142 83
.............
4 579 599 999 999 999 997 = 11*4163272727272727
4 579 599 999 999 999 998 = 2*121081*18911307306679
4 579 599 999 999 999 999 = 3*17*19*6625387*713333333
100 faktoriseringer beregnet paa:  333481.4427ms
 dvs:   3334.8144ms. per faktorisering

 largestLongFactorizedSafe: 4 579 599 841 640 001 173= 2139999949*2139999977
```

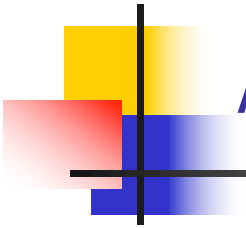
Om å parallelliser et problem

- Utgangspunkt: Vi har en sekvensiell effektiv og riktig sekvensiell algoritme som løser problemet.
- Vi kan dele opp både koden og data (hver for seg?)
- Vanligst å dele opp data
  - Som oftest deler vi opp data, og lar 'hele' koden virke på hver av disse data-delene (en del til hver tråd).
  - Eks: Matriser
    - radvis eller kolonnevis oppdeling av C til hver tråd
    - Omforme data slik at de passer bedre i cachene (transponere B)
  - Rekursiv oppdeling av data ('lett')
    - Eks: Quicksort
- Også mulig å dele opp koden:
  - Alternativ Oblig3 i INF1000: Beregning av Pi (3,1415..) med 17 000 sifre med tre ArcTan-rekker
  - Primtalls-faktorisering av store tall N for kodebrekking:
    - $N = p_1 * p_2$

Å dele opp algoritmen

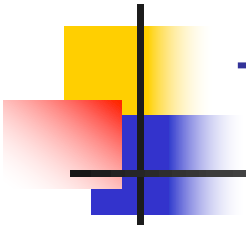


- Koden består en eller flere steg; som oftest i form av en eller flere samlinger av løkker (som er enkle, doble, triple..)
- Vi vil parallellisere med k tråder, og hver slikt steg vil få hver sin parallellisering med en CyclickBarrier-synkronisering mellom hver av disse delene + en synkronisert avslutning (join(), ..).
- Eks:
  - finnMax – hadde ett slikt steg: for (int i = 0 …n-1) -løkke
  - MatriseMult hadde ett slikt steg med trippel-løkke
  - Flere steg mulig: Eksempler senere i kurs (Radix)



Å dele opp data – del 2

- For å planlegge parallellisering av ett slikt steg må vi finne:
  - Hvilke data i problemet er lokale i hver tråd?
  - Hvilke data i problemet er felles/delt mellom trådene?
- Viktig for effektiv parallell kode.
  - Hvordan deler vi opp felles data (om mulig)
  - Kan hver tråd beregne hver sin egen, disjunkte del av data
  - Færrest mulig synkroniseringer (de tar 'mye' tid)

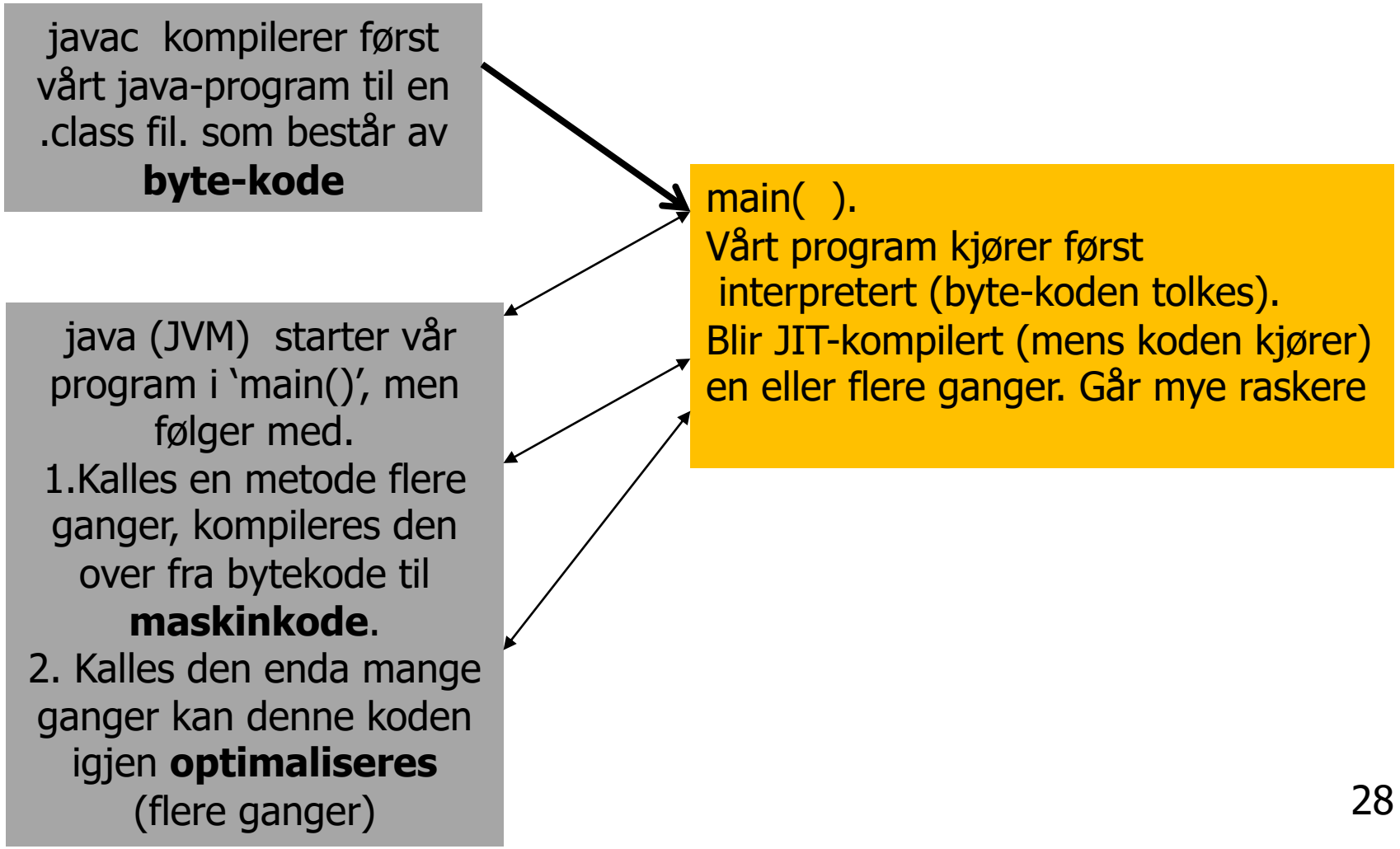


Tidtagning

- JIT –kompilering
  - Hvor mye betyr det egentlig
- Operativsystemet (Windows eller Linux)
  - Er de like raske?
- Søppeltømming i Java
  - Skjer under kjøring (med i tidene)

Tidsmålinger og JIT (Just In Time) -kompilering

- Tilbake til kompileringen av et Java-program:



Optimaliserng – ett eksempel

Original kode

```
class A {
  B b;
  public void newMethod() {
    y = b.get();
    ...do stuff...
    z = b.get();
    sum = y + z;
  }
}
class B {
   int value;
   final int get() {
      return value;
   }
}
```

1) Inline get

```
public void
newMethod() {
    y = b.value;
    ...do stuff...
    z = b.value;
    sum = y + z;
}
```

2) Fjern overflødige les

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    z = y;
    sum = y + z;
}
```

3) Fjern overflødige variable

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    y = y;
    sum = y + y;
}
```

4) Fjern død kode

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    sum = y + y;
}
```

Mediantider for finnMax fra ukeoppgavene:

n= 10 000

Vi ser at kjøretidene (sekv og para) synker dramatisk fra 1.ste til neste kjøring.
Pga JIT-optimalisering

```
M:\INF2440Para\FinnMax>java FinnMaxMulti 10000 7

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     6.30 msek. , nanosek/n:  630.46
Max sekv = a:9853, paa:     0.28 msek. , nanosek/n:   28.38

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.27 msek. , nanosek/n:   26.95

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.35 msek. , nanosek/n:   35.07
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    0.66

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.43 msek. , nanosek/n:   43.47
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.33

Kjøring:5, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.49 msek. , nanosek/n:   49.20
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:6, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.48 msek. , nanosek/n:   47.84
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.43

Median seq time:   0.014, median para time:   0.569,
 Speedup:    0.03, n = 10000
```

```
M:\INF2440Para\FinnMax>java FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    14.08 msek. , nanosek/n:    1.41
Max sekv = a:9999216, paa:     6.98 msek. , nanosek/n:    0.70

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     3.17 msek. , nanosek/n:    0.32
Max sekv = a:9999216, paa:     4.75 msek. , nanosek/n:    0.47

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.79 msek. , nanosek/n:    0.28
Max sekv = a:9999216, paa:     5.04 msek. , nanosek/n:    0.50

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.87 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.05 msek. , nanosek/n:    0.51

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.92 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.03 msek. , nanosek/n:    0.50

Median seq time:   5.052, median para time:   3.173,
 Speedup:    1.59, n = 10 000 000
```

n= 10 mill

```
M:\INF2440Para\FinnMax>java -Xint FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    67.24 msek. , nanosek/n:    6.72
Max sekv = a:9999216, paa:   179.40 msek. , nanosek/n:   17.94

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    64.00 msek. , nanosek/n:    6.40
Max sekv = a:9999216, paa:   175.12 msek. , nanosek/n:   17.51

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    51.42 msek. , nanosek/n:    5.14
Max sekv = a:9999216, paa:   176.23 msek. , nanosek/n:   17.62

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    64.95 msek. , nanosek/n:    6.49
Max sekv = a:9999216, paa:   173.17 msek. , nanosek/n:   17.32

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    60.11 msek. , nanosek/n:    6.01
Max sekv = a:9999216, paa:   185.84 msek. , nanosek/n:   18.58

Median seq time: 179.403, median para time:  64.950,
 Speedup:    2.76, n = 10 000 000
```

JIT-kompilering avslått :
> java –Xint
…..
n= 10 mill

```
M:\INF2440Para\FinnMax>java FinnM 100000000 5
Kjoering:0, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 41.913504 ms.
Max verdi sekvensiell i a:99989305, paa: 238.799921 ms.

Kjoering:1, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 26.78024 ms.
Max verdi sekvensiell i a:99989305, paa: 235.431219 ms.

Kjoering:2, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 27.791271 ms.
Max verdi sekvensiell i a:99989305, paa: 248.066478 ms.

Kjoering:3, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 26.86283 ms.
Max verdi sekvensiell i a:99989305, paa: 236.013201 ms.

Kjoering:4, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 27.755575 ms.
Max verdi sekvensiell i a:99989305, paa: 223.535073 ms.


Median sequential time:236.013201, median parallel time:27.755575,
 n=  100000000, Speedup:  8.59
```

n= 100 mill

JIT-kompilering +optimalisering

Søppel-tømming

Hva betyr dette for tidsmålingene

- Første gangen vi gjører er tiden vi måler en sum av:
  - Først litt interpretering av bytekode
  - Så oversetting(kompilering) av hyppig brukte metoder til maskinkode
  - kjøring av resten av programmet dels i maskinkode.
- Andre gang vi kjører, kan følgende skje:
  - JVM finner at noen av maskinkompilerte metodene våre må optimaliseres ytterligere
  - Kjøretiden synker ytterligere
- Tredje gang er som oftest optimaliseringa ferdig, men ytterligere optimalisering kan bli gjort
- Tidtakningen vår må endres !
- Vi kjører det sekvensielle og parallelle programmet f.eks 9 ganger i en løkke , noterer alle kjøretider i to arrayer som så sorteres og vi velger medianverdien = a[(a.length-1)/2]
- Du får aldri samme svaret to ganger – mye variasjon !!

FinnMax 3 ulike kjøringer (samme parametre , varierer antall tråder: 8, 16, 4)

Uke2>java FinnM 1000000 9
Kjøring:0**, ant kjerner:8, antTråder:8**
Max verdi parallell i a:999216, paa: 23.860968 ms.
Max verdi sekvensiell i a:999216, paa: 3.468803 ms.

Kjøring:1, ant kjerner:8, antTråder:8
Max verdi parallell i a:999216, paa: 0.311465 ms.
Max verdi sekvensiell i a:999216, paa: 0.549437 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:8
Max verdi parallell i a:999216, paa: 0.422752 ms.
Max verdi sekvensiell i a:999216, paa: 0.532639 ms.

Median sequential time:0.52004,
median parallel time:0.429051,
Speedup: **1.26,** n = 1000000

Uke2>java FinnM 1000000 9
Kjøring:0, **ant kjerner:8, antTråder:16**
Max verdi parallell i a:999216, paa: 18.808946 ms.
Max verdi sekvensiell i a:999216, paa: 3.558043 ms.

Kjøring:1, ant kjerner:8, antTråder:16
Max verdi parallell i a:999216, paa: 1.847439 ms.
Max verdi sekvensiell i a:999216, paa: 0.453898 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:16
Max verdi parallell i a:999216, paa: 0.502542 ms.
Max verdi sekvensiell i a:999216, paa: 0.471396 ms.

Median sequential time:0.509891,
median parallel time:0.646726,
Speedup: **0.90,** n = 1000000

Uke2>java FinnM 1000000 9
Kjøring:0, **ant kjerner:8, antTråder:4**
Max verdi parallell i a:999216, paa: 16.154151 ms.
Max verdi sekvensiell i a:999216, paa: 3.75507 ms.

Kjøring:1, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 1.280854 ms.
Max verdi sekvensiell i a:999216, paa: 0.520741 ms.

Kjøring:2, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 0.557136 ms.
Max verdi sekvensiell i a:999216, paa: 0.509191 ms.
.....
Kjøring:8, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 0.628527 ms.
Max verdi sekvensiell i a:999216, paa: 0.52354 ms.

Median sequential time:0.520741, median parallel time:0.628527,
Speedup: **0.88**, n = 1000000

«Aldri» samme resultatet to ganger

Uke2>java FinnM 1000000 9
ant kjerner:8, antTråder:8, n = 1mill

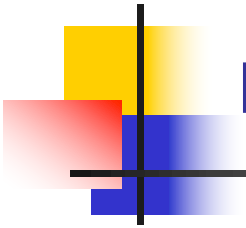
Med antall kjøringer for median = 9
1) Speedup: **0.68**, n = 1000000
2) Speedup: 0.96, n = 1000000
3) Speedup: 0.84, n = 1000000
4) Speedup: 0.71, n = 1000000
5) Speedup: 1.06, n = 1000000
6) Speedup: 1.26**,** n = 1000000

Med antall kjøringer for median = 21
7) Speedup: 1.00, n = 1000000
8) Speedup: 0.84, n = 1000000
9) Speedup: 0.88, n = 1000000
10) Speedup: **1.75**, n = 1000000
11) Speedup: 0.87, n = 1000000
12) Speedup: 1.11, n = 1000000
13) Speedup: 1.03, n = 1000000

Konklusjon på JIT-kompilering

- JIT-kompilering kan skrues av med >java –Xint MittProg ..
  - Brukes bare for debugging
- JIT kompilering kan gi 10 til 30 ganger så rask eksekvering for liten n (en god del mer for stor n)
- Første, andre (og tredje) kjøring er tidsmessig sterkt misvisende
- Vi må:
  - Kjøre programmet i en løkke f.eks 9 (eller 7 eller 11) ganger
  - Legge tidene i hver sin array (sekvensielt og parallell tid)
  - Sortere arrayene
  - Ta ut medianen (element (length-1)/2), som blir vår tidsmåling

```
import java.util.concurrent.*;
import java.util.*;
class Problem2 { int [] fellesData  ; // dette er felles, delte data for alle trådene
      double [] tidene ;
      int ant, svar;
      public static void main(String [] args) {
            ( new Problem()).utfoer(args);
      }
      void utfoer (String [] args) {
              ant = new Integer(args[0]);
              fellesData = new int [ant];
              tidene = new double[9];
              for (int m = 0; m <9; m++) {
                              long tid = System.nanoTime();
                              Thread t = new Thread(new Arbeider());
                               t.start();
                               try{t.join();}catch (Exception e) {return;}
                              tidene[m] = (System.nanoTime() -tid)/1000000.0;
                              System.out.println("Tid for "+m + ", tråd:"+tidene[m]+"millisec");
              }
              Arrays.sort(tidene);
              System.out.println("Median med svar:"+svar+", for trådene:"+tidene[(tidene.length-1)/2]+" millisec");
       } // end utfoer

       class Arbeider implements Runnable {
          int i,lokalData;    // dette er lokale data for hver tråd
                public void run() {     int sum =0;
                                        for (int i = 0; i < ant; i++) sum +=fellesData[i];
                                        svar =sum;
                }
    } // end indre klasse Arbeider
} // end class Problem
```



Hva med operativsystemet:

- Linux og Windows har om lag like rask implementasjon av Java og trådprogrammering,
- Dag Langmyhr testet to helt like maskiner med hhv. Linux og Windows, og resultatene tidsmessig (medianer) var nesten helt like, men
  - Ulike maskiner som Ifis store servere (diamant, safir,..) har en annen Linux og en noe langsommere ytelse for korte, trådbaserte programmer.

Hva med søppeltømming – garbage collection:

- Søppeltømming (=opprydding i lageret og fjerning av objekter vi ikke lenger kan bruke) kan slå til når som helst under kjøring:

```
Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.35 msek. , nanosek/n:   35.07
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    0.66

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.43 msek. , nanosek/n:   43.47
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.33

Kjøring:5, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.49 msek. , nanosek/n:   49.20
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36
```

IN3030 F08, våren 2023

Eric Jul
PT
Inst. for informatikk

Plan for F08

1. Prime Number Oblig: Questions?
2. Synchronization revisited
3. Cooks and waiters
4. 3 solutions
5. Hoare Monitors

Oblig 3 Prime Numbers

Questions?

Problemet vi nå skal løse: En restaurant med kokker og kelnere og med et varmebord hvor maten står

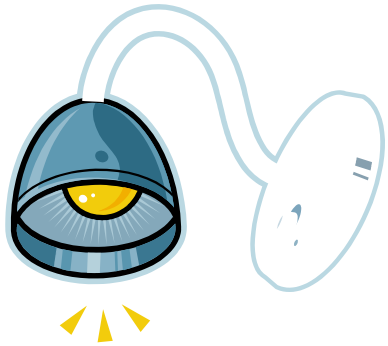
- Vi har **c** Kokker som lager mat og
  w Kelnere som server maten (tallerkenretter)
- Mat som kokkene lager blir satt fra seg på et **varmebord** (med TABLE_SIZE antall plasser til tallerkener)
- Kokkene kan ikke lage flere tallerkener hvis varmebordet er fullt
- Kelnerne kan ikke servere flere tallerkener hvis varmebordet er tomt
- Det skal lages og serveres NUM_TO_BE_MADE antall tallerkener

Restaurant versjon 1:

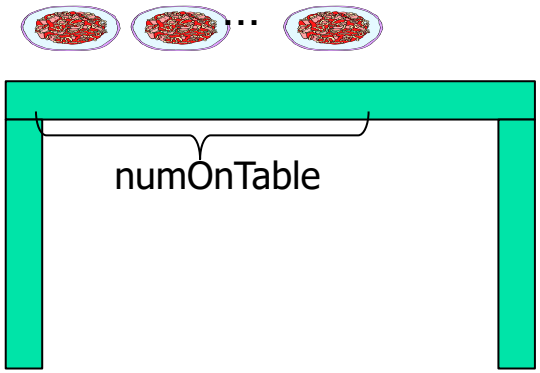
Lager tallerken-retter

c Kokker (produsenter)

varmelampe



TABLE_SIZE



Serverer tallerken

w Kelnere (konsumenter)

3) Om monitorer og køer (tre eksempler på concurrent programming). Vi løser synkronisering mellom to ulike klasser.

- **Først** en aktivt pollende (masende) løsning med synkroniserte metoder (Restaurant1.java).
  - Aktiv venting i en løkke i hver Kokk og Kelner
    + at de er i køen på å slippe inn i en synkronisert metode
- **Så** en løsning med bruk av monitor slik det var tenkt fra starten av i Java (Restaurant2.java).
  - Kokker og Kelnere venter i samme wait()-køen
    + i køen på å slippe inn i en synkronisert metode.
- **Til siste** en løsning med monitor med Lock og Condition-køer (flere køer – en per ventetilstand (Restaurant9.java)
  - Kelnere og Kokker venter i hver sin kø
    + i en køen på å slippe inn i de to metoder beskyttet av en Lock

.

Felles for de tre løsningene

```
import java.util.concurrent.locks.*;
class Restaurant {

  Restaurant(String[] args) {
     <Leser inn antall Kokker, Kelnere og  antall retter>
      <Oppretter Kokkene og Kelnerne og starter dem>
   }

    public static void main(String[] args) {
          new Restaurant(args);
       }
    } // end main
} // end class Restaurant

class HeatingTable{  // MONITOR
     int numOnTable  = 0,
         numProduced = 0,
         numServed=0;
     final int MAX_ON_TABLE  =3;
     final int NUM_TO_BE_MADE;
     // Invarianter:
     //   0 <= numOnTable  <=  MAX_ON_TABLE
     //          numProduced  <= NUM_TO_BE_MADE
     //          numServed      <= NUM_TO_BE_MADE

    < + ulike data i de tre eksemplene>
```

```
public xxx boolean putPlate(Kokk c)
   <Leggen tallerken til på bordet
   (true) ellers (false) Kokk må vente)
} // end put

public xxx boolean getPlate(Kelner w)  {
   <Hvis bordet tomt (false) Kelner venter
     ellers (true) - Kelner tar da en
     tallerken og serverer den>
  } // end get
} // end class HeatingTable

class Kokk extends Thread {
    HeatingTable tab;
    int ind;
    public void run() {
          while/if (tab.putPlate(..))
         < Ulik logikk i eksemplene>
    }
} // end class Kokk

class Kelner extends Thread {
    HeatingTable tab;
    int ind;
    public void run() {
          while/if (tab.getPlate()){
           <ulik logikk i eksemplene>
        }
} // end class Kelner
```

Invariantene på felles variable

- Invariantene må **alltid holde** (være sanne) utenfor monitor-metodene.
- Hva er de felles variable her:
  - MAX_ON_THE_TABLE
  - NUM_TO_BE_MADE
  - numOnTable
  - numProduced
  - numServed = numProduced – numOnTable
- Invarianter:
  1. **0 ≤ numOnTable ≤ TABLE_SIZE**
  2. **numProduced ≤ NUM_TO_BE_MADE**
  3. **numServed ≤ numProduced**

Invariantene viser 4 tilstander vi må ta skrive kode for

Invarianter:

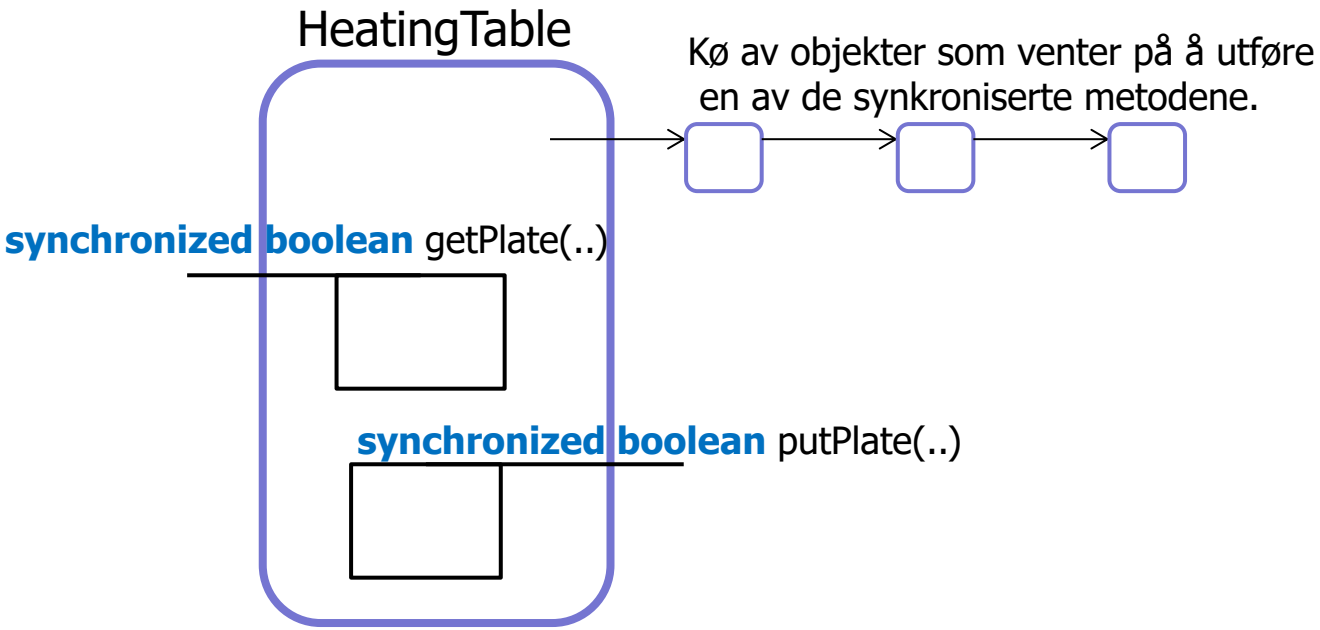
0 <= numOnTable <= MAX_ON_TABLE

numServed <= numProduced <= NUM_TO_BE_MADE

1. **numOnTable == MAX_ON_TABLE**
   → **Kokker venter**
2. **0 == numOnTable**
   → **Kelnere venter**
3. **numProduced == NUM_TO_BE_MADE**
   → **Kokkene ferdige**
4. **numServed == NUM_TO_BE_MADE**
   → **Kelnerene ferdige**

Først en aktivt pollende (masende) løsning med synkroniserte metoder (Restaurant1.java).

- Dette er en løsning med **en kø**, den som alle tråder kommer i hvis en annen tråd er inne i en synkronisert metode i samme objekt.
- Terminering ordnes i hver kokk og kelner (i deres run-metode)
- Den køen som nyttes er en felles kø av alle aktive objekter som evt. samtidig prøver å kalle en av de to synkroniserte metodene **get** og **put.** Alle objekter har en slik kø.



Restaurant løsning 1

```
class Kokk extends Thread {
....
    public void run() {
       try {
          while (tab.numProduced < tab.NUM_TO_BE_MADE) {
            if (tab.putPlate(this) ) {
               // lag neste tallerken
            }
            sleep((long) (1000 * Math.random()));
          }
       } catch (InterruptedException e) {}
       System.out.println("Kokk "+ind+" ferdig: " );
   }
} // end Kokk

class Kelner extends Thread {
.............
     public void run() {
          try {
            while ( tab.numServed< tab.NUM_TO_BE_MADE) {
                 if ( tab.getPlate(this)) {
                    // server tallerken
                 }
                 sleep((long) (1000 * Math.random()));
              }
          } catch (InterruptedException e) {}
            System.out.println("Kelner " + ind+" ferdig");
     }
} // end Kelner
```

```
synchronized boolean  putPlate(Kokk c) {
      if (numOnTable == TABLE_SIZE) {
                return false;
       }
       numProduced++;
      // 0 <= numOnTable  <  TABLE_SIZE
      numOnTable++;
      // 0 < numOnTable  <=  TABLE_SIZE
      System.out.println("Kokk no:"+c.ind+",
        laget  tallerken no:"+numProduced);
     return true;
} // end putPlate

synchronized boolean  getPlate(Kelner w) {
     if  (numOnTable == 0)     return false;
     // 0 < numOnTable  <=  TABLE_SIZE
     numServed++;
     numOnTable--;
     // 0 <= numOnTable <  TABLE_SIZE
     System.out.println("Kelner no:"+w.ind+
     ",  serverte  tallerken no:"+numServed);
    return true;
 }
```

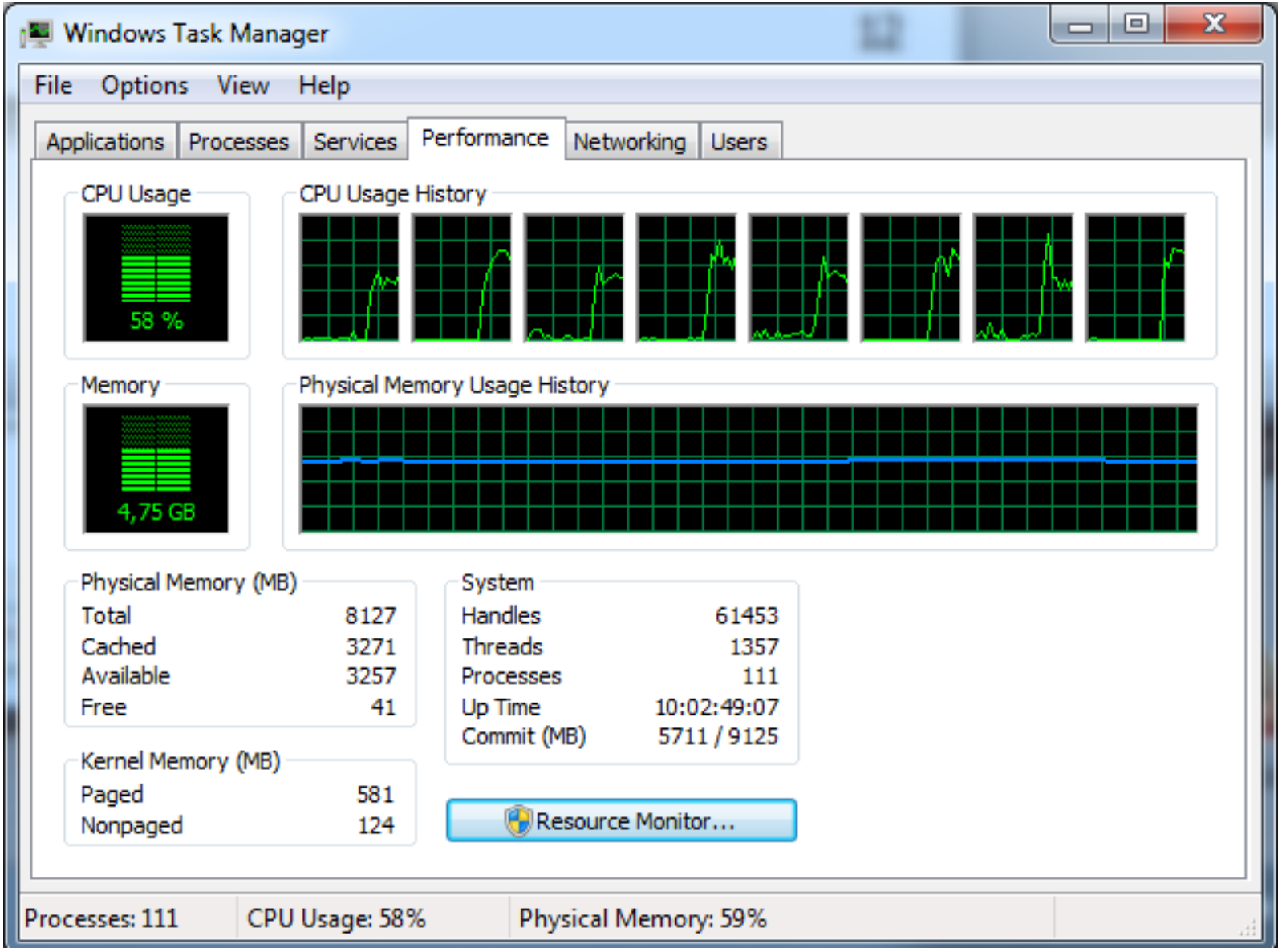
```
M:Restaurant1>java Restaurant1 11 8 8
Kokk no:8, laget  tallerken no:1
Kokk no:4, laget  tallerken no:2
Kokk no:6, laget  tallerken no:3
Kelner no:5, serverte  tallerken no:1
Kokk no:3, laget  tallerken no:4
Kelner no:2, serverte  tallerken no:2
Kokk no:1, laget  tallerken no:5
Kelner no:5, serverte  tallerken no:3
Kelner no:4, serverte  tallerken no:4
Kokk no:7, laget  tallerken no:6
Kokk no:2, laget  tallerken no:7
Kelner no:7, serverte  tallerken no:5
Kokk no:4, laget  tallerken no:8
Kelner no:3, serverte  tallerken no:6
Kelner no:3, serverte  tallerken no:7
Kokk no:1, laget  tallerken no:9
Kelner no:2, serverte  tallerken no:8
Kokk no:6, laget  tallerken no:10
Kokk no:3, laget  tallerken no:11
Kokk 8 ferdig:
```

```
Kelner no:8, serverte  tallerken no:9
Kelner no:7, serverte  tallerken no:10
Kelner no:6, serverte  tallerken no:11
Kokk 3 ferdig:
Kokk 5 ferdig:
Kelner 1 ferdig
Kokk 1 ferdig:
Kokk 4 ferdig:
Kelner 5 ferdig
Kokk 7 ferdig:
Kelner 2 ferdig
Kokk 2 ferdig:
Kelner 4 ferdig
Kelner 3 ferdig
Kelner 7 ferdig
Kelner 6 ferdig
Kokk 6 ferdig:
Kelner 8 ferdig
```

Problemer med denne løsningen er aktiv polling

- Alle Kokke- og Kelner-trådene går aktivt rundt å spør:
  - Er der mer arbeid til meg? Hviler litt, ca.1 sec. og spør igjen.
  - Kaster bort mye tid/maskininstruksjoner.
- Spesielt belastende hvis en av trådtypene (Produsent eller Konsument) er klart raskere enn den andre,
  - Eks . setter opp 18 raske Kokker som sover bare 1 millisek mot 2 langsomme Kelnere som sover 1000 ms.
  - I det tilfellet tok denne aktive ventingen/masingen 58% av CPU-kapasiteten til 8 kjerner
- Selv etter at vi har testet i run-metoden at vi kan greie en tallerken til, må vi likevel teste på om det går OK
  - En annen tråd kan ha vært inne og endret variable
- Utskriften må være i get- og put-metodene. Hvorfor?

Løsning1 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser maskinen med stadige mislykte spørsmål hvert ms. om det nå er plass til en tallerken på varmebordet . CPU-bruk = 58%.



Løsning 2: Javas originale opplegg med monitorer og **to køer**.

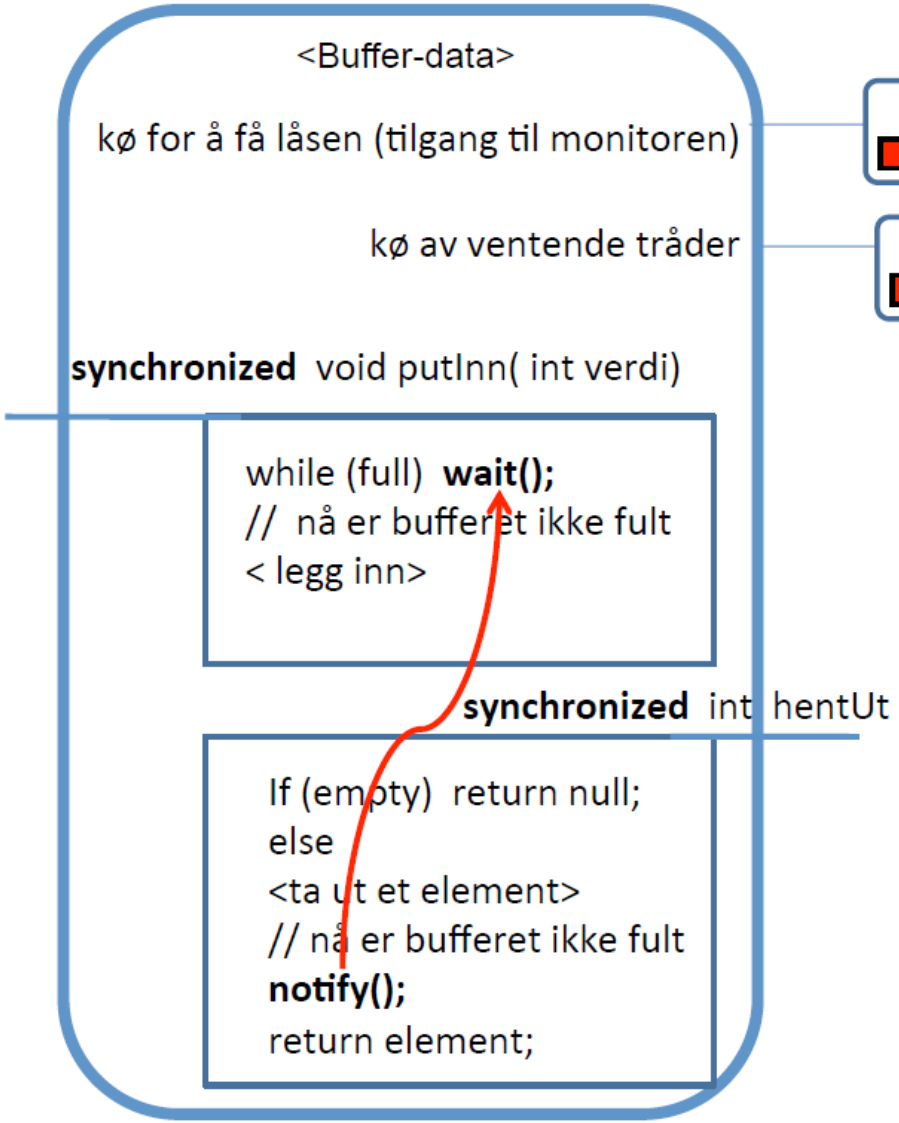
- Den originale Java løsningen med synkroniserte metoder og en rekke andre metoder og følgende innebygde metoder:
- **sleep(t):** Den nå kjørende tråden sover i 't' millisek.
- **notify():** (arvet fra klassen Object som alle er subklasse av). Den vekker opp **en** av trådene som venter på låsen i inneværende objekt. Denne prøver da en gang til å få det den ventet på.
- **notifyAll():** (arvet fra klassen Object). Den vekker opp **alle de** trådene som venter på låsen i inneværende objekt. De prøver da alle en gang til å få det de ventet på.
- **wait():** (arvet fra klassen Object). Får nåværende tråd til å vente til den enten blir vekket med notify() eller notifyAll() for dette objektet.

Å lage parallelle løsninger med en Java 'monitor'

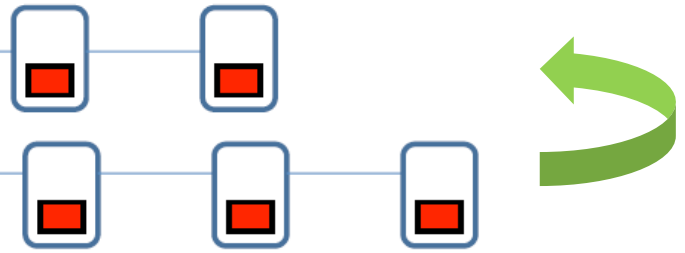
- En Java-monitor er et objekt av en vilkårlig klasse med synchronized metoder
- Det er to køer på et slikt objekt:
  - En kø for de som venter på å komme inn/fortsette i en synkronisert metode
  - En kø for de som her sagt **wait()** (og som venter på at noen annen tråd vekker dem opp med å si notify() eller notifyAll() på dem)
    - wait() sier en tråd inne i en synchronized metode.
    - notify() eller notifyAll() sies også inne i en synchronized metode.

Monitor-ideen er sterkt inspirert av Tony Hoare (mannen bak Quicksort)

To køer i en basal Java monitor:



En kø av ventende tråder på hele monitoren



notify()

En kø av ventende tråder på ”wait”-instruksjoner (wait-set).

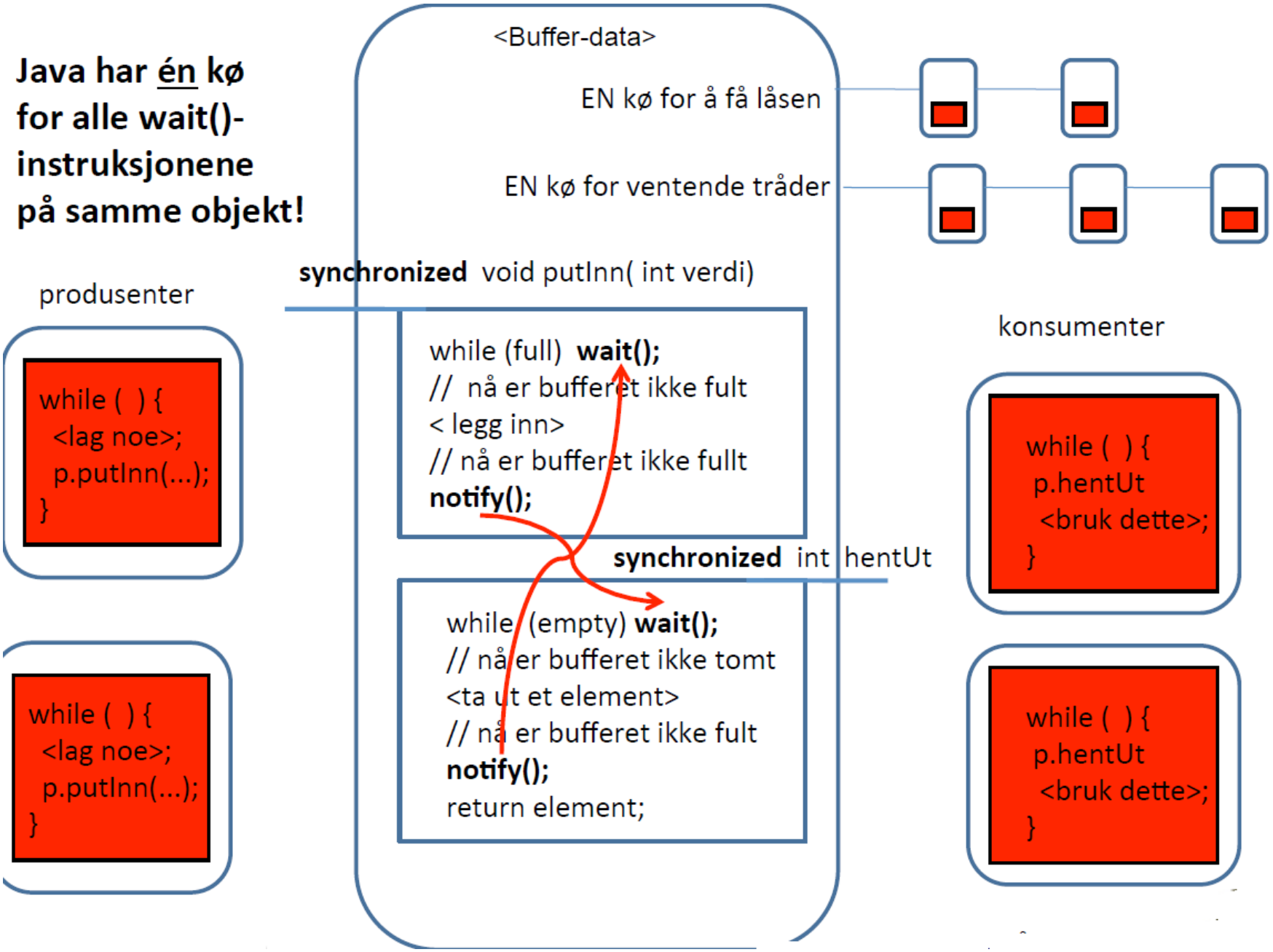
Startes av notify () og/eller notifyAll()

Legges da i den andre køen

(først ? (Nei, ingen garanti))

Derfor er det nødvendig med ”while …”

Java har <u>én</u> kø for alle wait()-instruksjonene på samme objekt!

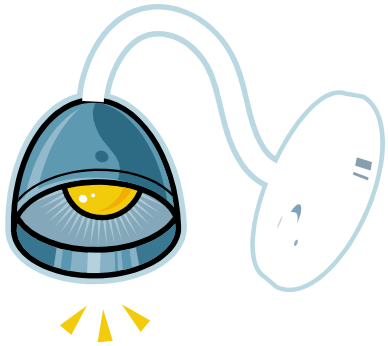


Pass på å unngå vranglås

Restauranten (2):

numProduced ≤ NOS_TO_BE_MADE

c Kokker (producers)



MAX_ON_THE_TABLE

numOnTable

numServed = = numProduced - numOnTable

w Kelnere (consumers)

En kø: Kokker og Kelnere venter samme wait-køen

Løsning 2, All venting er inne i synkroniserte metoder i en to køer.

- All venting inne i to synkroniserte metoder
- Kokker and Kelnere venter på neste tallerken i wait-køen
- Vi må vekke opp alle i wait-køen for å sikre oss at vi finner en av den typen vi trenger (Kokk eller Kelner) som kan drive programmet videre
- Ingen testing på invariantene i run-metodene

Begge løsninger 2) og 3):
run-metodene prøver en gang til hvis siste operasjon lykkes:

Kokker:

```
public void run() {
        try {
            while (tab.putPlate(this)) {
                sleep((long) (1000 * Math.random()));
            }
        } catch (InterruptedException e) {}
        // Denne Kokken er ferdig
    }
```

Kelnere:

```
public void run() {
        try {
            while (tab.getPlate(this) ){
                sleep((long) (1000 * Math.random()));
            }
        } catch (InterruptedException e) {}
          // Denne Kelneren er ferdig
    }
```

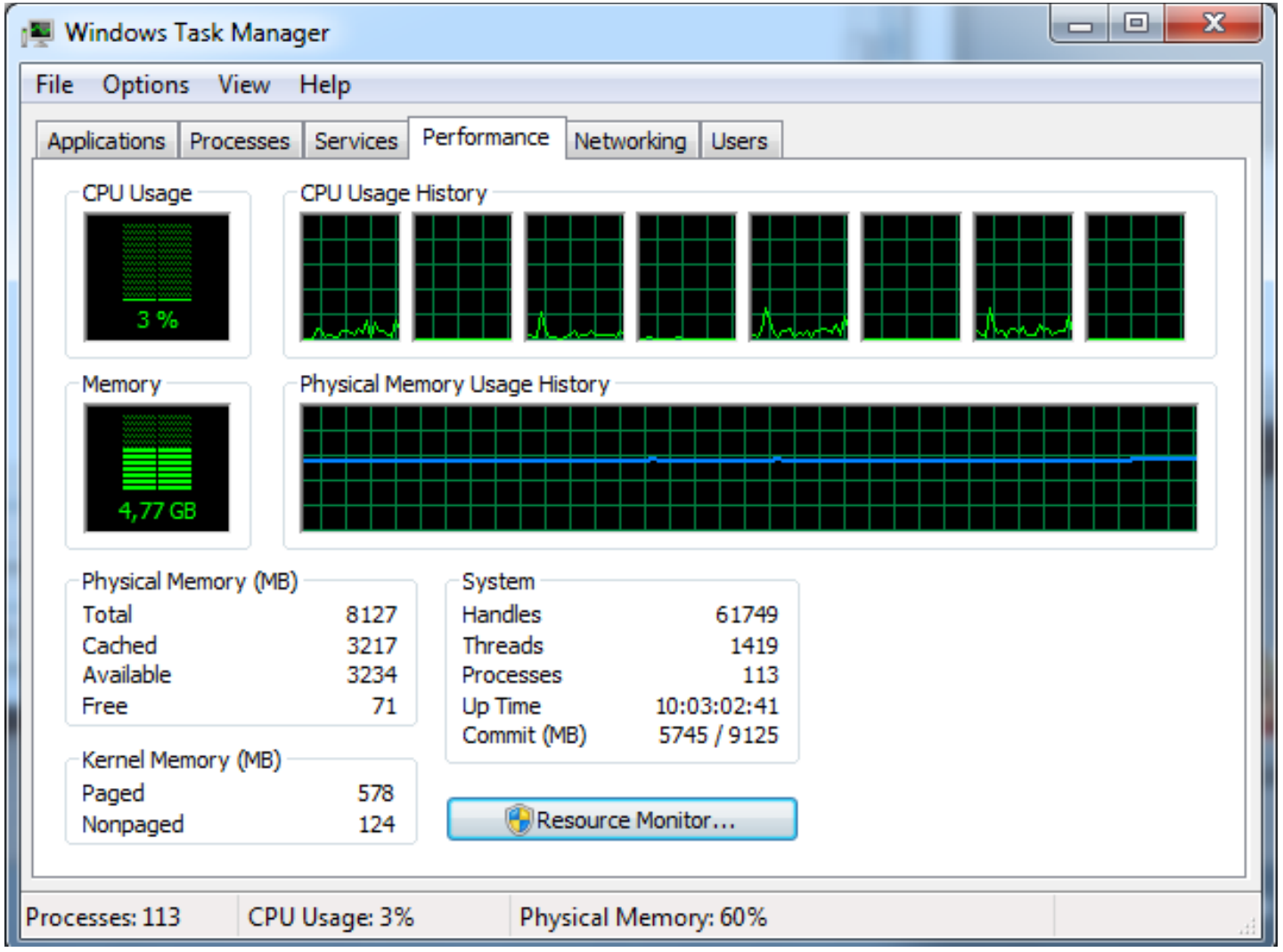
Løsning2

```
public synchronized boolean putPlate (Kokk c) {
    while (numOnTable == TABLE_SIZE &&
           numProduced <   NUM_TO_BE_MADE) {
        try { // The while test holds here meaning  that a Kokk should
              // but can not make a dish, because the table is full
            wait();
        } catch (InterruptedException e) { // Insert code to handle interrupt }
    }
    // one or both of the loop conditions are now false
    if (numProduced < NUM_TO_BE_MADE) {
        // numOnTable < TABLE_SIZE
        // Hence OK to increase numOnTable
        numOnTable++;
        // numProduced < NUM_TO_BE MADE
        // Hence OK to increase numProduced:
        numProduced++;
        // numOnTable > 0 , Wake up a waiting
        // waiter, or all if
        //numProduced ==NUM_TO_BE_MADE
        notifyAll();  // Wake up all waiting
        if (numProduced ==
            NUM_TO_BE_MADE) {
            return false;
        } else{ return true; }
    } else {
        // numProduced ==
        //NUM_TO_BE_MADE
        return false;}
} // end putPlate
```

Løsning2

```
public synchronized boolean getPlate (Kelner w) {
      while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
            try { // The while test holds here the meaning that the table
                  // is empty and there is more to serve
                  wait();
            } catch (InterruptedException e) { // Insert code to handle interrupt }
       }
      //one or both of the loop conditions are now false
      if (numOnTable > 0) {
            // 0 < numOnTable <= TABLE_SIZE
            // Hence OK to decrease numOnTable:
            numOnTable--;
            // numOnTable < TABLE_SIZE
            // Must wake up a sleeping Kokker:
            notifyAll();   // wake up all queued Kelnere and Kokker
            if (numProduced == NUM_TO_BE_MADE && numOnTable == 0) {
                  return false;
            }else{ return true;}
      } else { // numOnTable == 0 && numProduced == NUM_TO_BE_MADE
            return false;}
} // end getPlate
```

Løsning2 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser ikke maskinen med stadige mislykte spørsmål , men venter i kø til det er plass til en tallerken til på varmebordet . CPU-bruk = 3%.



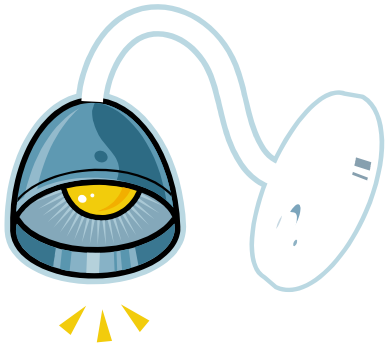
3) En parallell løsning med Conditions.

- Bruker to køer:
  - En for Kokker som venter på en tallerkenplass på bordet
  - En for Kelnere som venter på en tallerken
- Da trenger vi ikke vekke opp alle trådene, **Bare en** i den riktige køen.
  - Kanskje mer effektivt
  - Klart lettere å programmere

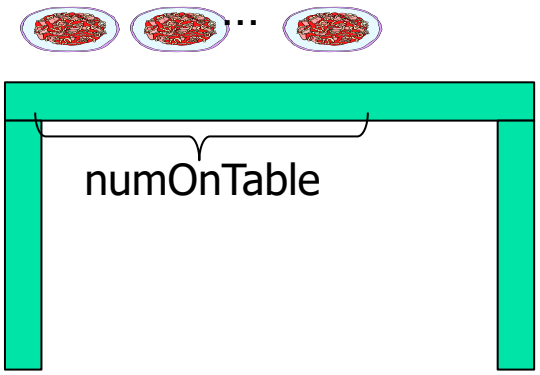
Restauranten (3):

numProduced ≤ NUM_TO_BE_MADE

c Kokker (produsenter)



MAX_ON_THE_TABLE



numServed = = numProduced – numOnTable

w Kelnere (konsumenter)

To køer:
a) Kokker venter på en plass på bordet

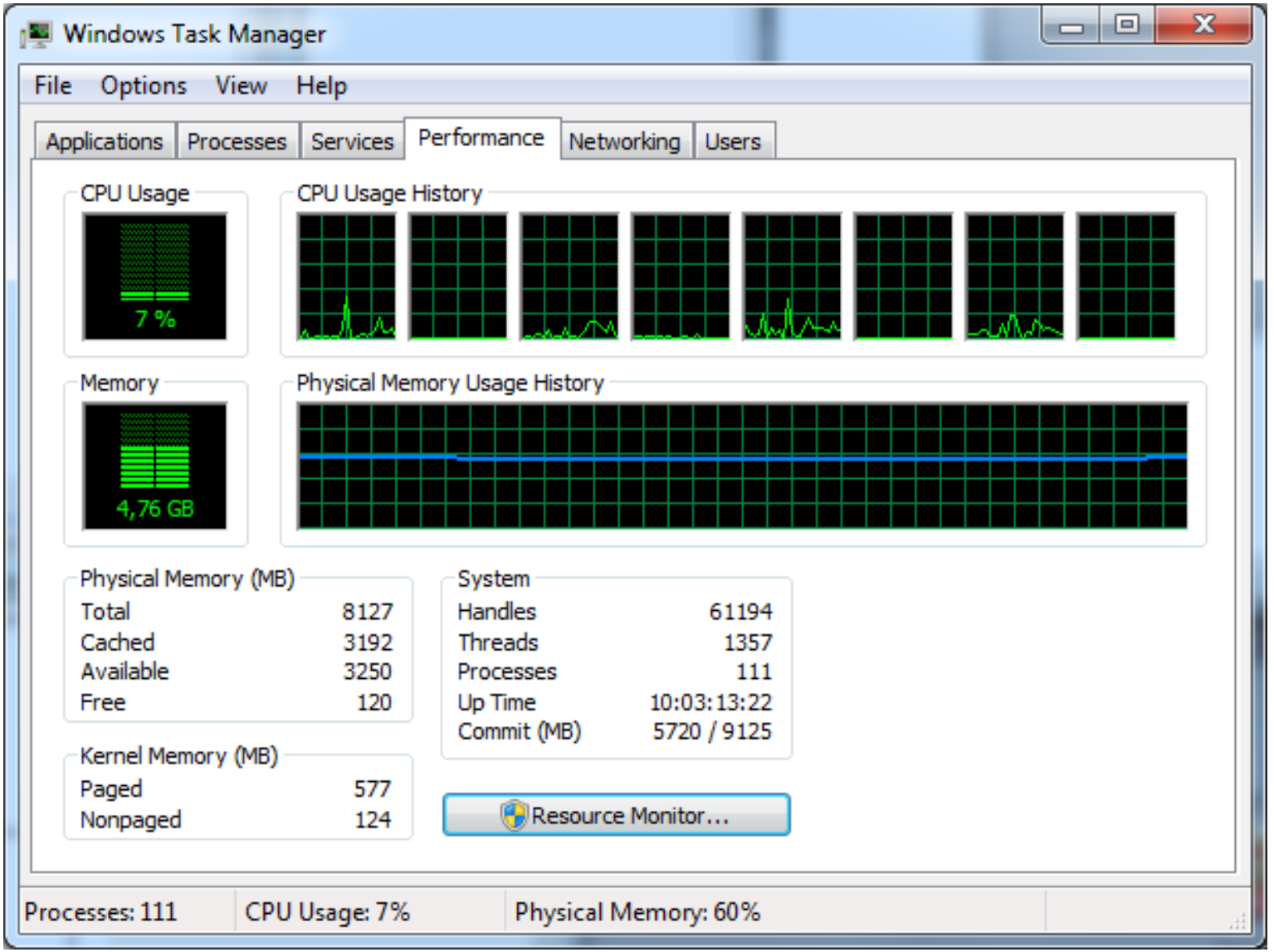
b) Kelnere venter på flere tallerkener

```
final Lock lock = new ReentrantLock();
final Condition notFull  = lock.newCondition();     // kø for Kokker
final Condition notEmpty = lock.newCondition(); // kø for Kelnere

public boolean putPlate (Kokker c) throws InterruptedException {
     lock.lock();
      try {while (numOnTable == MAX_ON_TABLE && numProduced < NUM_TO_BE_MADE){
                notFull.await(); // waiting for a place on the table
             }
            if (numProduced < NUM_TO_BE_MADE) {
                  numProduced++;
                  numOnTable++;;
                  notEmpty.signal(); // Wake up a waiting Kelner to serve
                  if (numProduced == NUM_TO_BE_MADE) {
                        //  I have  produced the last plate,
                       notEmpty.signalAll();   // tell Kelnere to stop waiting, terminate
                      notFull.signalAll();       // tell Kokker to stop waiting and terminate
                       return false;
                   }
                 return true;
             } else {  return false;}
      } finally {
         lock.unlock();
  }  } // end putPlate
```

```
public boolean getPlate (Kelnerw) throws InterruptedException {
    lock.lock();
    try {
        while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
            notEmpty.await();      // This Kelner is waiting for a plate
        }
        if (numOnTable > 0) {
            numOnTable--;
            notFull.signal();  // Signal to one Kokk in the Kokker's waiting queue
            return true;
        } else {
            return false;}
    } finally {
        lock.unlock();
    }
} // end getPlate
```

En Kelner eller en Kokk blir signalisert av to grunner:
- for å behandle (lage eller servere) en tallerken til
- ikke mer å gjøre, gå hjem (tøm begge køene)

Løsning3 med 18 raske Kokker (venter 1 ms) og 2 langsomme Kelnere (venter 1000 ms). Kokkene stresser ikke maskinen med stadige mislykte spørsmål , men venter i kø til det er plass til en tallerken til på varmebordet . CPU-bruk = 7%.



Vurdering av de tre løsningene

- **Løsning 1**: Enkel, men kan ta for mye av CPU-tiden. Særlig når systemet holder av andre grunner å gå i metning vil typisk en av trådene da bli veldig treige, og da tar denne løsningen plutselig ½-parten av CPU-tiden.
- **Løsning 2:** God, men vanskelig å skrive
- **Løsning 3:** God, nesten like effektiv som løsning 2 og lettere å skrive.

Avsluttende bemerkninger til Produsent-Konsument problemet

- Invarianter brukes av alle programmerere (ofte ubevisst)
  - program, loop or metode (sekvensiell eller parallell)
  - Å si dem eksplisitt hjelper på programmeringen
- HUSK: synchronized/lock virker bare når alle trådene synkroniserer på samme objektet.
  - Når det skjer er det **sekvensiell tankegang** mellom wait/signal
- Når vi sier notify() eller wait() på en kø, vet vi ikke:
  - Hvilken tråd som starter
  - Får den tråden det er signalisert på kjernen direkte etter at den som sa notify(), eller ikke ??. Ikke definert
- Debugging ved å spore utførelsen (trace) – System.out.println(”..”)
  - Skriving utenfor en Locket/synkronisert metode/del av metode, så lag en:
    - synchronized void println(String s) {System.out.println(s);}
  - Ellers kan utskrift bli blandet eller komme i gal rekkefølge.

systems out of smaller functions, which are separately packaged and interconnected. The availability of large functions, combined with functional design and construction, should allow the manufacturer of large systems to design and construct a considerable variety of equipment both rapidly and economically.

Linear circuitry

Integration will not change linear systems as radically as digital systems. Still, a considerable degree of integration will be achieved with linear circuits. The lack of large-value capacitors and inductors is the greatest fundamental limitations to integrated electronics in the linear area.

By their very nature, such elements require the storage of energy in a volume. For high Q it is necessary that the volume be large. The incompatibility of large volume and integrated electronics is obvious from the terms themselves. Certain resonance phenomena, such as those in piezoelectric crystals, can be expected to have some applications for tuning functions, but inductors and capacitors will be with us for some time.

The integrated r-f amplifier of the future might well consist of integrated stages of gain, giving high performance at minimum cost, interspersed with relatively large tuning elements.

Other linear functions will be changed considerably. The matching and tracking of similar components in integrated structures will allow the design of differential amplifiers of greatly improved performance. The use of thermal feedback effects to stabilize integrated structures to a small fraction of a degree will allow the construction of oscillators with crystal stability.

Even in the microwave area, structures included in the definition of integrated electronics will become increasingly important. The ability to make and assemble components small compared with the wavelengths involved will allow the use of lumped parameter design, at least at the lower frequencies. It is difficult to predict at the present time just how extensive the invasion of the microwave area by integrated electronics will be. The successful realization of such items as phased-array antennas, for example, using a multiplicity of integrated microwave power sources, could completely revolutionize radar.